SOFTWARE ARCHITECTURE

Edited by

Patrick Donohoe

IFIP

SPRINGER SCIENCE+
BUSINESS MEDIA, LLC

SOFTWARE ARCHITECTURE

IFIP - The International Federation for Information Processing

IFIP was founded in 1960 under the auspices of UNESCO, following the First World Computer Congress held in Paris the previous year. An umbrella organization for societies working in information processing, IFIP's aim is two-fold: to support information processing within its member countries and to encourage technology transfer to developing nations. As its mission statement clearly states,

IFIP's mission is to be the leading, truly international, apolitical organization which encourages and assists in the development, exploitation and application of information technology for the benefit of all people.

IFIP is a non-profitmaking organization, run almost solely by 2500 volunteers. It operates through a number of technical committees, which organize events and publications. IFIP's events range from an international congress to local seminars, but the most important are:

- The IFIP World Computer Congress, held every second year;
- open conferences;
- working conferences.

The flagship event is the IFIP World Computer Congress, at which both invited and contributed papers are presented. Contributed papers are rigorously refereed and the rejection rate is high.

As with the Congress, participation in the open conferences is open to all and papers may be invited or submitted. Again, submitted papers are stringently refereed.

The working conferences are structured differently. They are usually run by a working group and attendance is small and by invitation only. Their purpose is to create an atmosphere conducive to innovation and development. Refereeing is less rigorous and papers are subjected to extensive group discussion.

Publications arising from IFIP events vary. The papers presented at the IFIP World Computer Congress and at open conferences are published as conference proceedings, while the results of the working conferences are often published as collections of selected and edited papers.

Any national society whose primary activity is in information may apply to become a full member of IFIP, although full membership is restricted to one society per country. Full members are entitled to vote at the annual General Assembly, National societies preferring a less committed involvement may apply for associate or corresponding membership. Associate members enjoy the same benefits as full members, but without voting rights. Corresponding members are not represented in IFIP bodies. Affiliated membership is open to non-national societies, and individual and honorary membership schemes are also offered.

SOFTWARE ARCHITECTURE

**TC2 First Working IFIP Conference on Software Architecture (WICSA1)
22-24 February 1999, San Antonio, Texas, USA**

Edited by

Patrick Donohoe
Carnegie Mellon University

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

ISBN 978-1-4757-6536-6 ISBN 978-0-387-35563-4 (eBook)
DOI 10.1007/978-0-387-35563-4

Library of Congress Cataloging-in-Publication Data

A C.I.P. Catalogue record for this book is available
from the Library of Congress.

Printed on acid-free paper.

The original version of the book frontmatter was revised: The copyright line was incorrect. The Erratum to the book frontmatter is available at DOI: 10.1007/978-0-387-35563-4_35

Contents

Organizing Committee

General conference chair
Paul Clements, *Software Engineering Institute, Carnegie Mellon University, USA*

Program co-chairs
Dewayne E. Perry, *Bell Laboratories, USA*
Alexander Ran, *Nokia Research Center, USA*

IFIP TC-2 chair
Reino Kurki-Suonio, *Tampere University of Technology, Finland*

Program Committee

Bruce Anderson, *IBM, UK*
Bob Balzer, *USC-ISI, USA*
Don Batory, *University of Texas at Austin, USA*
Judy Bishop, *University of Pretoria, South Africa*
Maarten Boasson, *Hollandse Signaalapparaten BV, The Netherlands*
Manfred Broy, *Technical University of Munich, Germany*
José L. Fiadeiro, *University of Lisbon, Portugal*
David Garlan, *Carnegie Mellon University, USA*
Morven Gentleman, *Institute for Information Technology, National Research Council, Canada*
Carlo Ghezzi, *Politecnico di Milano, Italy*
K. Gopinath, *Indian Institute of Science, India*
Volker Gruhn, *University of Dortmund, Germany*
Christine Hofmeister, *Siemens Corporate Research, USA*
Ric Holt, *University of Waterloo, Canada*
Paola Inverardi, *Università dell'Aquila, Italy*
Mehdi Jazayeri, *University of Vienna, Austria*
Takuya Katayama, *Japan Advanced Institute of Science and Technology, Japan*
Rick Kazman, *Software Engineering Institute, Carnegie Mellon University, USA*
Tomoji Kishi, *NEC Corporation, Japan*
Jeff Kramer, *Imperial College, UK*
Philippe Kruchten, *Rational Software, Canada*
Juha Kuusela, *Nokia Research Center, Finland*
Ora Lassila, *Nokia Research Center, USA*

Daniel Le Metayer, *INRIA/IRISA, Campus de Beaulieu, Rennes, France*
Jeff Magee, *Imperial College, UK*
Naftaly Minsky, *Rutgers University, USA*
Jürgen Müller, *Philips Research Laboratories, The Netherlands*
Henk Obbink, *Philips Research Laboratories, The Netherlands*
Frances Paulisch, *Siemens AG, Germany*
Franklin Reynolds, *Nokia, USA*
David Rosenblum, *University of California at Irvine, USA*
Mary Shaw, *Carnegie Mellon University, USA*
Sylvia Stuurman, *Delft University of Technology, The Netherlands*
Will Tracz, *Lockheed Martin, USA*
Hong Zhu, *Software Institute, Nanjing University, PRC*

Preface

Software architecture is a primary factor in the creation and evolution of virtually all products involving software. It is a topic of major interest in the research community where promising formalisms, processes, and technologies are under development. Architecture is also of major interest in industry because it is recognized as a significant leverage point for manipulating such basic development factors as cost, quality, and interval. Its importance is attested to by the fact that there are several international workshop series as well as major conference sessions devoted to it.

The First Working IFIP Conference on Software Architecture (WICSA1) provided a focused and dedicated forum for the international software architecture community to unify and coordinate its effort to advance the state of practice and research. WICSA1 was organized to facilitate information exchange between practising software architects and software architecture researchers. The conference was held in San Antonio, Texas, USA, from February 22nd to February 24th, 1999; it was the initiating event for the new IFIP TC-2 Working Group on Software Architecture.

This proceedings document contains the papers accepted for the conference. The papers in this volume comprise both experience reports and technical papers. The proceedings reflect the structure of the conference and are divided into six sections corresponding to the working groups established for the conference.

Patrick Donohoe
Software Engineering Institute
Carnegie Mellon University

ANALYSIS AND ASSESSMENT OF SOFTWARE ARCHITECTURE

Architecture Design Recovery of a Family of Embedded Software Systems

An Experience Report

Lars Bratthall and Per Runeson
Dept. of Communication Systems, Lund University, Sweden.
P.O. Box 118, S-221 00 Lund, Sweden.
Phone: +46-462229668., Fax +46-46145823. Email {lars.bratthall|per.runeson}@tts.lth.se.

Key words: Architectural design recovery, experience report, qualitative evaluation

Abstract: Understandability of the current system is a key issue in most reengineering processes. An architecture description of the system may increase its understandability. This paper presents experiences from architectural design recovery in a product family of large distributed, embedded systems. Automated recovery tools were hard to apply due to the nature of the source code. A qualitative evaluation procedure was applied on the performance of the recovery process. The results suggest that producing the necessary architectural documentation during the recovery project costs eight to twelve times as much as producing the same set of documentation during the original development project. By applying a common architectural style for all members of the product family, the component reuse made possible decreased source code volume by 65%.

1. INTRODUCTION

A part of any reengineering project is to create an understanding of the architecture of the current system. This understanding can help determine which pieces are reusable, and to what extent. Also, the current architecture can pose requirements on later developed systems (Abowd et al., 1997). Documentation of the software architecture may also decrease the large proportion of time maintainers spent on developing an understanding of the entity to modify (Holtzblatt et al., 1997). In this paper we present experiences from a project where architectural level design recovery was performed on a product family of five distributed, embedded, software systems.

Design recovery is a phase in reverse engineering where source code and external knowledge are used to create abstractions beyond those obtained directly by examining the system itself (Chikofsky and Cross II, 1990). Biggerstaff (1989) argues that "Design recovery must reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth." In the project studied, the available source models (Murphy and Notkin, 1995) were the source code for a product family and a few pages of documentation. The access to original system experts was very limited. It was not known what quality attributes the architecture of the software possessed, except that it executed well. It was not known whether the members of the product family shared any common software architecture. The hardware was however well described and identical for all members of the product family. The source code was spread over 90 to 150 files for each member of the product family.

An incremental approach to recovering information from the source code was adopted. To simplify future maintenance the architectural style "Layers" (Shaw and Garlan, 1996) was imposed, due to its known quality properties (maintainability aspects). Imposing an architecture was believed to be feasible as a recovered architecture can be considered an interpretation of a less abstract entity. Different tools for architectural design recovery were investigated, but due to performance constraints only tools that operated on static code could be used. Automated analysis has been discussed by several authors e.g.,, Chase et al. (1998), Harris et al. (1996) and Holtzblatt et al. (1997). Due to certain constructs frequently used in the source code examined, the value of these methods was considered limited.

The software architecture was recovered largely by hand using simple tools like *grep* and *emacs*. SDL (ITU-T, 1996a) was used as architecture description language. Once the architecture of one member of the product family had been recovered, this architecture was reused when attempting architectural recovery on other members of the product family. With some restructuring and minimal reengineering (Chikofsky and Cross II, 1990), both component reuse and architecture reuse (Karlsson, 1995) were used, resulting in a common architecture for all members of the product family as well as a reduction of the total code volume by 65%.

2. CONTEXT

The studied system was contracted to Ericsson Microwave Systems AB who develops complex systems. One of their product areas is telecommunications. The studied project aimed at designing a family of switches. The switches shared the same set of hardware components, except for different special-purpose printed circuits. One family of subsystems within the switches was studied.

For various reasons the software was not documented according to existing quality standards; the only existing source models available to maintainers were 300 000 lines of C source code, some assembler, and a few pages of documentation, the latter giving little clue regarding the architecture. This rendered any kind of maintenance difficult, as long time had to be allocated just to understand code. Future architectural erosion (Perry and Wolf, 1992) was feared, as there was no known rationale for the architectural design decisions taken.

In order to solve these problems, an architectural design recovery project was launched.

3. OVERVIEW OF THE ARCHITECTURAL DESIGN RECOVERY PROJECT

Biggerstaff (1989) describes a general design recovery process with maintenance and the population of a reuse library as objectives. In this paper, the focus is on practical experiences gained in applying this process.

Biggerstaff's process has three steps:

1. Supporting program understanding,
2. Supporting population of reuse and recovery libraries, and
3. Applying the outcomes of design recovery for refining the recovery.

These steps are applied iteratively.

3.1 Step 1 — Program understanding for maintenance

An architecture recovery team needs some initial knowledge. It includes:

- Details of the available source models
- Available design recovery tools
- Knowledge of what code to allocate to different components.

These issues were addressed initially.

3.1.1 Details of the available source models

Examining the make files showed that some of the files were never used. Examining the filenames showed similarity in the filenames between different members of the product family, and usually the contents of files with the same filename were similar to some extent. Closer examination indicated that what had originated as identical files had eroded to slightly different files. The analysis also showed that identical C functions sometimes were allocated to different files, without any obvious rationale.

3.1.2 Investigation of design recovery tools

A number of tools believed to be beneficial in design recovery were investigated. Results indicated that a semi-manual approach was needed.

Making a call graph did not help very much, since the subsystems were based on concurrent software processes, communicating mainly using the real-time operating system built-in signals. The call graph showed intra-process communication fairly well, but inter-process communication was not described well.

Identification of a signal being sent could be automated; simple *grep* commands can look for operating system keywords used to create and send signals. Identification of the receiving software process for signals was difficult; we could not rely on pure lexical analysis, since the receiver of a signal usually was determined at run time. Dynamic analysis by executing the system on the target-system could possibly have provided input to event trace analysis (Jerding and Rugaber, 1997), but we were unable to automatically create event traces due to certain constructs frequently used:

- Other mechanisms than signals were sometimes used, especially direct read/write to memory. This communication could not be traced without impeding the function of the system due to performance violations.
- Communication to other subsystems was handled using signals wrapped into special-purpose packets. The operating system debugger could not symbolically show the contents of these packets.

Further tool support was not investigated. Dynamic analysis conflicted with performance requirements, while automatic static recovery tools would have trouble handling the distributed nature, the special-purpose packets, the usage of direct memory read/ write, and the dynamic determination of receiving software processes. Thus, we in many cases had to identify the receiver of signals by manually walking through scenarios (well defined dynamic sequences).

3.1.3 Code to allocate to components

Some source files belonged to only one software process, while some files needed restructuring as parts of the code in one file belonged to more than one software process. There were also two COTS (Commercial Off-The-Shelf) products involved (the operating system and a TCP/IP stack), each spread across a set of files.

The design artefacts to recover were a static architectural description, interwork descriptions, and different dynamic models.

3.2 Step 2 — Populating reuse and recovery libraries

Based on the input from step 1, a set of hypotheses was decided on.

- Manual work during step 2 and step 3 would be necessary, since a recovered software architecture is an interpretation, not entirely visible in code (Holtzblatt et al., 1997).
- Software processes would be the initial abstraction level of the software components. Thus we used a variant of Harris et al.'s (1996) approach, that equated components with software processes. After looking at code, it was found that trying to divide software processes into smaller components, e.g., concurrent state machines, would be difficult as we could not distinguish the individual state machines in the software processes. Therefore we choose software processes as the initial abstraction level.
- Component connectors were to be represented by inter-process signalling. The contents of inter-subsystem communication packets were to be tracked rather than the special-purpose packet itself. Function calls inside a software process would not be described, since we estimated that recovering this information would be too much work related to the use a maintainer would have.
- Describing the architecture of a member of the product family by showing all software processes and their data/control connectors would show too much detail in some situations. Aggregated as well as non-aggregated components should be provided. The smallest component would consist of code related to a single software process.
- Simple tools like *grep* and *emacs* would be the main tools for analysis. SDL would be used to represent the static architecture description. Message Sequence Charts (ITU-T, 1996b) would be used to represent the control and data flow between components.
- For project reasons, an incremental approach allowing the premature termination and later continuation of the architectural recovery was needed.

This led to the workflow described in table 1. On the horizontal axis, activities performed are shown. On the vertical axis, levels of increased value of the recovered artefacts are shown. Components are created at increasing abstraction levels, named C_1, C_2 and C_3. Level C_n components are aggregated from level C_{n-1} components.

Table 1. Goals versus performed activities

	A. Identify software processes	B. Allocate source files to software processes, abstraction level C_1	C. Restructure source files	D. Create special components	E. Cluster level C_1 components into level C_2 components	F. For every software process: Identify sent signals and receivers	G. Describe signals	H. Clarify service provider/requester	I. Cluster level C_2 components into level C_3 components	J. Represent using an ADL
Baseline established	A									
Source code allocated to level C_1 components		B	C							
COTS components handled				D						
Level C_2 components defined					E	F	G			
Level C_3 components defined								H	I	
Architecture graphically described										J

3.2.1 Creation of first order components (level C_1) - activities A-D

All source files belonging to a software process were assigned to one C_1 component. All assembler files were allocated to one C_1 component. Each set of COTS files was allocated to one C_1 component each.

Some files could not be associated with a single software process despite restructuring. These functions were assigned to a library component. The types of level C_1 components created were Single Software Process components, Library components, Assembler components and COTS components.

Level C_1 components were fairly easy to identify; simple tools allowed partly automated analysis. As the source code was not very interleaved (Rugaber et al., 1995) only little restructuring was needed.

3.2.2 Creation of second order components (level C_2) - activities E-G

In order to raise the component abstraction level from each component containing only one software process, to components containing several such components an iterative approach was used. A graphical representation of inter-C_1 control and data communication was drawn using SDL. The inter-process communication constructs prior identified in the source were represented by SDL signals or SDL remote procedure calls. By analysing the communication routes, the type and amount of communication, level C_2 components were decided on. If a set of level C_1 components solved one easily delimited task, they were to be clustered into a level C_2 component.

Identifying level C_2 components was more difficult than identifying level C_1 components. Exact rules for clustering could never be devised, since some level C_1 components participated in solving more than one task.

3.2.3 Creation of third order components (level C_3) - activities H-J

The source code indicated that there were similarities between the members of the product family. We attempted to impose a layered architectural style (Shaw and Garlan, 1996), by clarifying service provider/requester relationships between components. Some restructuring of the original C_2 components was required.

Grouping of level C_2 components into C_3 layer components was done by looking at 'distance from hardware'. All hardware-close level C_2 components were assigned to a level C_3 layer component 'Hardware Abstraction Layer'. Other level C_3 layer components, with decreasing knowledge of hardware specifics, were 'Subsystem Controller', 'Main Controller' and 'Supervision and Test'.

There were several reasons for attempting the layered architectural style:

- The layered architectural style is well known for its good maintainability properties.
- By dividing hardware-close functionality from control, we expected greater chances of component reuse in other members of the product family.

We expected to be able to decrease the difference between different members of the product family by using a common architectural style for all of them.

This multiple-level component architecture was represented in SDL. SDL was chosen, as it allows the direct representation of architectural features (Harris et al., 1996) such as software processes, components consisting of one or more processes, aggregated components, components without any software process, inter-process signalling and remote procedure calls. Thus, many issues related to the representation problem (Rugaber and Clayton 1993) were avoided. However, there was some semantic distance between C and SDL that had to be mapped: Direct memory reads/writes, interrupts, and the special-purpose packet used to convey signals between different subsystems. These constructs were mapped to SDL signals and SDL remote procedure calls. We used naming conventions to distinguish these constructs from the direct mapping between C source signals and these other communication constructs.

3.3 Step 3 — Applying the outcomes of the design recovery

The above steps were applied for one member of the product family. In doing design recovery for the other members of the product family, the already defined components and the architectural style were reused. By restructuring components by merging files if possible, the number of new components was held down.

Design recovery for the other products in the family was much quicker than for the first member. A large part of the improvement came from having to document few new components. Also, knowing the expected architectural style, less work had to be done in choosing how to restructure the software to fit the architecture.

The degree of reusability of components was proportional to the distance from hardware. The closer to hardware, the more easily could components be reused. By having several component abstraction levels (C_3, C_2, C_1), we could reuse parts or whole of components:

- Most level C_1 hardware-close components could be reused at least once.
- Some members of the product family could share level C2 components.
- Some members of the product family could share level C3 components. A few new level C3 components had to be created, usually by replacing only a few level C_1 components inside a level C_3 component.

The layered architectural style could be reused for all members of the product family.

4. RESULTS AND EXPERIENCES GAINED

The project resulted in a common architectural style for all members of the product family. This enabled component reuse, that decreased the total code volume (lines of source code) by 65%. The volume of architectural descriptions and component descriptions were reduced by approximately 30%, relatively what would have been needed if no reuse had been applied.

A number of faults were discovered in the process of comparing components from different members of the product family.

The set of hypothesis described in section 3.2 remained unmodified through-out the project. However, they would probably have changed if the first step in Biggerstaff's process had not been. This first step helped in deciding on the set of work hypotheses.

It is a daunting task to do architectural recovery when tools can provide only limited aid. Subjective estimation indicates that the effort of our recovery/reuse project amounted to eight to twelve times the effort to accomplish the same results (architectural description, common architectural style, component-based architecture) during the original development project. The estimation is based on accurate figures for the recovery/restructure project and subjective estimations regarding how to handle the problem during original development. Future maintenance is expected to be much simpler and faster than would be possible without the architectural descriptions and component design. Without the design recovery project any maintenance would be extremely difficult.

Experiences have been collected by conducting interviews with the designers involved in the architecture recovery project, as well as future maintainers and some involved managers. Experiences reported are related to tools, people and the recovery process used.

4.1 Tool support

Tools have been used for recovery as well as representation of the recovered architecture. During recovery, UNIX *grep* and the colour marking functions of *emacs* were helpful, especially combined into small scripts. *Grep* allowed the searching of common features across several members of the product family. *Emacs* helped in performing manual slicing, as well as it helped in comparing several versions of files automatically.

SDL has shown to be suitable for describing the component architecture down to the software process level. It was possible to unambiguously describe the constructs believed usable for our purposes. Some semantic distance demanded mapping rules between C and SDL. We believe SDL to be a possible architecture description language for systems, where components are mainly based on software processes and connectors are mainly inter-process communication.

A future challenge to solve is that there is no automatic correspondence between the source code and the architectural abstractions. For example, lack of intense communication between two components is not necessarily a sign of that the two components should not be aggregated into a larger component. In the system studied, this was apparent when we decided to group hardware-close components into an aggregated component.

4.2 People

Experiences related to people concern previous knowledge and other intellectual instruments for design recovery. As the rationale for architectural decisions is not seen in C, having even limited access to original designers have been extremely beneficial. They have been able to provide information that has not been available in other source models.

Having knowledge of architectural styles helped in choosing to use a layered architecture, as well as trying to establish the service provider/requester divisions, which is a client/server architectural style (Shaw and Garlan, 1996). It is believed that any recovery team can benefit from having access to original design knowledge, domain architecture knowledge and knowledge of architectural styles. Manual design recovery is error prone. This emphasises the need for automated design recovery, or better yet, do it right during the original development.

4.3 The recovery process

Dynamic analysis was difficult due to performance issues. For the purpose of maintenance, dynamic models are considered necessary. Better original descriptions would have been preferred, or, an elaborate debugging component should have been available. For example, being able to run the software on the target platform with relaxed timing requirements would have aided in analysing the software dynamically.

The interleaving problem was rarely encountered, as we never split software processes into more than one component. Content coupling, in terms of several processes sharing a library of functions, was handled by either restructuring those files (by splitting them and allocating them to separate components) or allocating the library functions to a separate library component. By dividing the recovery process into discrete steps, management gained visibility into the project and could decide on project alterations and resource allocation. The incremental approach was thus perceived as beneficial.

5. CONCLUSIONS AND THE FUTURE

From the studied architecture recovery project, we conclude that the design recovery process described by Biggerstaff (1989) works, but undertaking a design recovery project with limited access to system experts and other source models than the source code, is a daunting task. Especially, understanding hardware-close software is difficult, as it requires detailed hardware understanding. Knowledge of architectural styles and their properties help in choosing a suitable architecture to represent the code, as one knows what quality attributes a particular architecture possesses. An incremental approach to recovering the software architecture is beneficial since it increases visibility into the recovery process.

The recovery project would have benefited from a larger set of well-defined component connectors. Full semantics for the mapping between source code and an architecture description language would allow the automatic creation and simultaneous maintenance of code and architectural views.

Tool support for architectural recovery is important. In industrial projects like this, where the product is supposed to have a life-span of at least 15 years, any description of the architecture should be represented using commercially available tools. We agree with researchers, e.g., Kazman and Carrière (1998), claiming that several methods are necessary in a design recovery project, thus concluding that a workbench with open interfaces is a suitable architecture for design recovery tools.

ACKNOWLEDGEMENTS

This work was partly funded by The Swedish National Board for Industrial and Technical Development (NUTEK), grant 1K1P-97-09690. The project was conducted while employed at the Q-Labs Group. Employees at Ericsson Microwave Systems AB and members of the Software Engineering Research Group at the Department of Communication Systems, Lund University, have provided insightful input.

REFERENCES

Abowd, G., Goel, A., Jerding, D.F., McCracken, M., Moore, M., Murdock, J.W., Potts, C., Rugaber, S., Wills, L. (1997) MORALE. Mission ORiented Architectural Legacy Evolution, in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society, Los Alamitos, USA, 150–9.

Biggerstaff, T.J. (1989) Design Recovery for Maintenance and Reuse. *IEEE Computer*, 22(7), 36–49.

Chase, M.P., Christey, S.M., Harris, D.R., Yeh, A. S. (1998) Recovering Software Architecture from Multiple Source Code Analyses, in *Proceedings of the ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering*.

Chikofsky, E.J., Cross II, J.H. (1990) Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1), 13–7.

Harris, D.R., Yeh, A.S., Reubenstein, H.B. (1996) Extracting Architectural Features from Source Code. *Automated Software Engineering*, 3(1/2), 109–38.

Holtzblatt, L.J., Piazza, R.L., Reubenstein, H.B., Roberts, S.N., Harris, D.R. (1997) Design Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, 23(7), 461–72.

ITU-T (1996a) *Recommendation Z.100. Specification and Description Language, SDL*, International Telecommunication Union.

ITU-T (1996b) *Recommendation Z.120. Message Sequence Charts*, International Telecommunication Union.

Jerding, D., Rugaber, S. (1997) Using Visualization for Architectural Localization and Extraction, in *Proceedings of the Fourth Working Conference on Reverse Engineering*, IEEE Computer Society, Los Alamitos, USA, 56–65.

Karlsson, E.A. (1995) *Software Reuse - A Holistic Approach*. John Wiley, Chichester, Great Britain.

Kazman, R., Carrière, S.J. (1998) View Extraction and View Fusion in Architectural Understanding, in *Proceedings of the Fifth International Conference on Software Reuse*, IEEE Computer Society, Los Alamitos, USA, 290–9.

Murphy, G.C., Notkin, D. (1995) Lightweight Source Model Extraction. *SIGSOFT Software Engineering Notes*, 20(4), 116–27.

Perry, D.E., Wolf, A.L. (1992) Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), 40–52.

Rugaber, S., Clayton, R. (1993) The Representation Problem in Reverse Engineering, in *Proceedings of Working Conference on Reverse Engineering*, IEEE Computer Society, Los Alamitos, USA, 8–16.

Rugaber, S., Stirewalt, K., Wills, L.M. (1995) The Interleaving Problem in Program Understanding, in *Proceedings of the Second Working Conference on Reverse Engineering*, IEEE Computer Society, Los Alamitos, USA, 166–75.

Shaw, M. and Garlan, D. (1996) *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, New Jersey, USA.

A Software Architecture Reconstruction Method

George Yanbing Guo[1], Joanne M. Atlee[1] & Rick Kazman[2]
[1]Department of Computer Science, University of Waterloo, Waterloo, ON, Canada
[2]Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, U.S.A
yguo@se.math.uwaterloo.ca, jmatlee@dragon.uwaterloo.ca, rkazman@sei.cmu.edu

Key words: Design recovery, reverse engineering, software architecture analysis, design patterns, pattern recognition

Abstract: Changes to a software system during implementation and maintenance can cause the architecture of a system to deviate from its documented architecture. If design documents are to be useful, maintenance programmers must be able to easily evaluate how closely the documents conform to the code they are meant to describe. Software architecture recovery, which deals with the extraction and analysis of a system's architecture, has gained more tool support in the past few years. However, there is little research on developing effective and efficient architectural conformance methods. In particular, given the increasing emphasis on patterns and styles in the software engineering community, a method needs to explicitly aid a user in identifying architectural patterns.

This paper presents a semi-automatic method, called ARM (Architecture Reconstruction Method), that guides a user in the reconstruction of software architectures based on the recognition of patterns. Once the system's actual architecture has been reconstructed, we can analyze conformance of the software to the documented design patterns.

1. INTRODUCTION

A *software architecture* is a high-level description of a software system's design, often a model of the software's components (e.g., objects, processes, data repositories, etc.), the externally visible properties of those components, and the relationships among them (Bass, et al., 1998). The concept of software architectures has received considerable attention lately, and developers are starting to document software architectures. However, the living architecture of a software system may drift from the documented architecture if architecture changes are made during software implementation or maintenance and no similar effort is made to maintain the architecture documents. Although architectural integrity could, in theory, be enforced by a rigorous review process, in practice this is seldom done.

Architecture conformance analysis can be used to evaluate how well the architecture of a software system corresponds to its documentation; it can also assist in keeping the architecture documents up to date. Some progress on this problem has been made at the source file and module levels, where the software's call-graph is extracted from source code and compared with the expected call-graph (Murphy, et al., 1995), (Woods & Yang, 1995). In addition, a number of reverse engineering tools have been developed to automatically extract, manipulate, and query source model information (e.g., REFINE (Reasoning, -). Imagix (Imagix, -), Rigi (Wong, et al., 1994), (Storey, et al., 1996), LSME (Murphy & Notkin, 1996), IAPR (Kazman & Burth, 1998), RMTool (Murphy, et al., 1995)).

Design patterns are an attempt to codify solutions to recurring problems, to make routine design easier. In an architecture, design patterns prescribe specific abstractions of data, function, and interconnections. Automated conformance analysis of newer software architectures is actually complicated by the use of design patterns and architectural styles in architecture documents. While this statement seems at first to be contradictory to the thesis of this paper, the complication stems from the fact that extraction tools extract *code-level* information, not architectural information. Hence, the analyst needs some way to map from the low-level extracted information up to architectural concepts. To properly analyze the architectures of systems developed using design patterns, we need tools and techniques for recognizing instances of pattern-level abstractions.

This paper shows how code-level extraction can feed into pattern-based architecture conformance analysis. We present a semi-automatic analysis method, called ARM (Architecture Reconstruction Method), that codifies heuristics for applying existing reverse-engineering tools (for reasoning about code-level artifacts) to the problem of recognizing more abstract patterns in the implementation. Once the system's actual architecture has been reconstructed, we can analyze conformance of the software to the documented design patterns.

Following this introduction, section 2 provides a review of software architecture recovery. Section 3 describes ARM in detail. Evaluation of the method with case studies is presented in section 4. Finally, section 5 summarizes this work and proposes future research.

2. SOFTWARE ARCHITECTURE RECOVERY

Software architecture recovery can be divided into two phases:

1. identification and extraction of source code artifacts, including the architectural elements; and
2. analysis of the extracted source artifacts to derive a view of the implemented architecture.

The extracted source artifacts form a *source model*, which comprises a collection of *elements* (e.g., functions, files, variables, objects, etc.), a set of *relations* between the elements (e.g., "function calls function", "object A has an instance") and a set of *attributes* of these elements and relations (e.g., "function calls function N times"), to represent the system (Kazman & Carriere, 1998).

2.1 Architecture recovery frameworks

There exist many source model extraction tools, such as LSME (Murphy & Notkin, 1996), SNiFF+ (SniFF, -), ManSART (Yeh, et al., 1997) and Imagix (Imagix, -), that parse code fragments and extract source model elements, relations and attributes. Tools that use relational algebra to infer new facts from existing facts, such as SQL and Grok (Holt, 1998), can be used to manipulate and analyze source model artifacts. Tools for extracting and analyzing software architectures, such as Rigi (Wong, et al., 1994), CIA (Chen, et al., 1990) and SAAMTool (Kazman, 1996), provide not only visualization but also manipulation mechanisms to help the user simplify and navigate through the visual system representation. However, each individual tool or system has its limitations and restrictions in terms of the architecture recovery phases it covers, its support for applications developed in different programming languages and its flexibility in supporting customized analysis.

A *software architecture framework* integrates and leverages multiple tools in an organized structure to facilitate architecture recovery.

Kontogiannis et al. have developed a toolset, called RevEngE (Reverse Engineering Environment), to integrate heterogeneous tools, such as Ariadne (Kontogiannis, et al., 1994), ART (Johnson, 1993) and Rigi (Wong, et al., 1994) for extracting, manipulating and analyzing system facts, through a common repository specifically designed to support architecture recovery.

The architecture recovery framework of the Software Bookshelf project (Finnigan, et al., 1997) provides access to a variety of extractors, such as C Fact Extractor (CFX) and CIA, for source model extraction. Manipulation and analysis of the source model stored in the repository is possible via tools like grep, sort, or Grok, to emit architectures of the subsystems and of the system. The architecture that Bookshelf produces is a hierarchical structural decomposition of system in terms of subsystems, files, and functions. The architectures can be visualized using tools such as the Landscape Viewer.

The *Dali* architecture workbench (Kazman & Carriere, 1999), is an infrastructure for the integration of a wide variety of extraction, manipulation, analysis, and presentation tools. The architecture recovery work presented in this paper was performed using Dali.

2.2 The Dali workbench

Dali's architecture is shown in Figure 1, where the rectangles represent distinct tools and lines represent data flow among them.

Source model extraction can be performed by a variety of lexical-based, parser-based or profiling-based tools that produce static or dynamic *views* of the system under examination. A *view* is a source model extracted by a single extraction tool or technique. A static view contains static source artifacts extracted from source code. A dynamic view contains dynamic elements including dynamic typing information, process spawning and instances of interprocess communication (IPC). These extracted views are stored in a repository, currently a relational database. The various extracted views can be fused together into *fused views* (Kazman & Carriere, 1998).

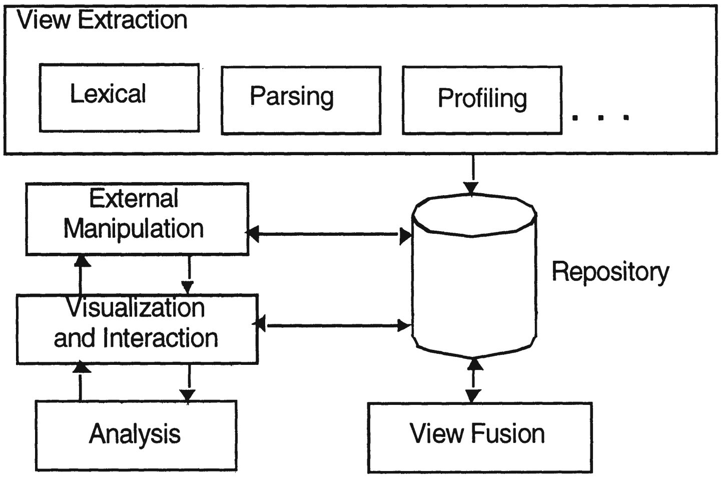


Figure 1: The Dali workbench

Visualization tools can be deployed in Dali to present the source model and the result of architecture analysis. For example, Rigi is used to present systems as a graph with nodes denoting the artifacts and arcs representing the relations between them. Dali supports various external manipulation and analysis tools, such as Grok, IAPR (Kazman & Burth, 1998), and RMTool (Murphy, et al., 1995). The system view can be exported to these tools and the analysis results can be added back to the repository. Using Rigi's command language, new tools can be added in Dali and a software analyst can choose among tools when performing an analysis task. Dali does not rely on having an Abstract Syntax Tree (AST). This allows it to cope with architecture analysis on systems that can not be parsed.

2.3 Architecture recovery methods

Automated tools and frameworks can be used to extract and reason about code-level facts. However, human input is needed to extract and infer facts about higher-level abstractions (e.g., design patterns). An *architecture recovery method* defines a series of steps, and the pre/post conditions for each step, to guide an analyst in systematically applying existing reverse engineering tools to recover a system's architecture.

Most current architecture recovery methods are based on a system decomposition hierarchy to reason about software architecture by looking at the relations (calls and uses relations in most cases) between the subsystems, between the files and between the functions (Portable Bookshelf, -). However, it is difficult to use these methods to recover architectures that are designed and implemented with design patterns. As design patterns are described as well-defined structures with constraint rules, a pattern-oriented architecture recovery method must incorporate the design pattern rules as well as structural information such as the system decomposition hierarchy.

Shull et al. developed the BACKDOOR analysis method to recognize design patterns in object-oriented systems (Schull, et al., 1996). This method uses a general abstract pattern description, rather than an application-specific pattern instantiation, to guide pattern recognition, and hence could be ineffective in producing accurate results. The pattern definition, detection and evaluation in this method are performed manually, which makes the method primarily applicable to small systems.

3. ARCHITECTURE RECONSTRUCTION METHOD

To assist software architecture recovery of systems designed and developed with patterns, we developed the Architecture Reconstruction Method (ARM)—a semi-automatic analysis method for reconstructing architectures based on the recognition of architectural patterns.

ARM is depicted in Figure 2. As indicated by the dashed boxes in this figure, ARM consists of four major phases:

1. Developing a concrete pattern recognition plan.
2. Extracting a source model.
3. Detecting and evaluating pattern instances.
4. Reconstructing and Analyzing the architecture.

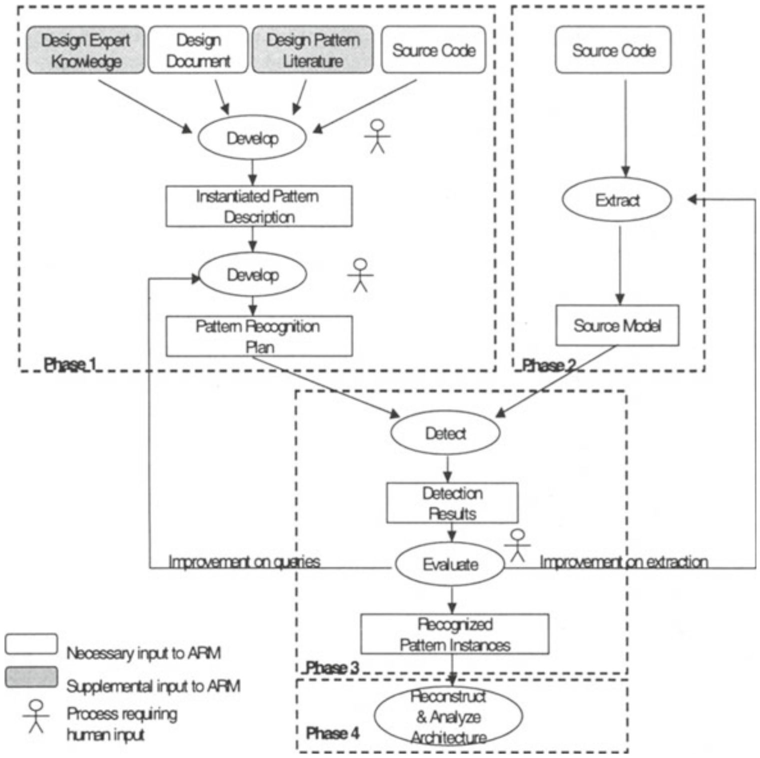


Figure 2: Pattern recognition process flow chart

3.1 Developing a concrete pattern recognition plan

Constructing a pattern recognition plan consists of three steps. The first is to develop an instantiated pattern description. By *instantiation*, we mean a concrete pattern description, with all the pattern elements and their relations described in terms of the constructs available from the chosen implementation language. Starting with a design document, one can manually determine the patterns used in the design and can extract the *abstract pattern rules*—the design rules that define a pattern's structural and behavioral properties. Pattern descriptions found in the design pattern literature, e.g., (Buschmann, et al., 1996), or obtained from humans who are familiar with the system design can be used to supplement these rules. Using these abstract pattern rules as a guide, one can then examine the source code of several potential pattern instances to derive the corresponding *concrete pattern rules*—the implementation rules that realize abstract pattern rules using data structures, coding conventions, coding methods and algorithms. Such concrete pattern rules can be recognized via syntactic cues, such as naming conventions and programming language keywords, or an analysis of data access and control flow.

An instantiated pattern description is a specification of the concrete pattern rules written in Rigi Standard Format (RSF) (Wong, et al., 1994). A clause in RSF is a tuple (relation, entity1, entity2), which represents the relationship *entity1 relates to entity2*. For example in the *Mediator design pattern* (see Figure 3), a *Mediator* component serves as the communication hub for all the *Colleague* components. An abstract pattern rule for this pattern is

"The Mediator component mediates communications between colleague components."

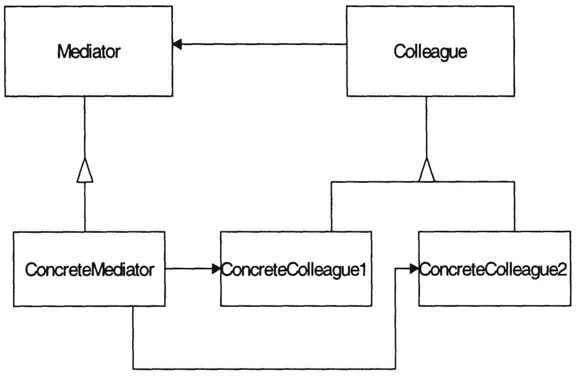


Figure 3: Mediator design pattern

In one of our case studies, the *Mediator pattern* is implemented in a C++ class where mediator and colleague components are member functions. [1]

[1] This use of the mediator design pattern is an adaptation of what is found in (Gamma, et al., 1994).

Based on call sequence information (control flow), the following concrete pattern rule is identified to realize the above abstract rule where function B is a Mediator and function A and C are Colleagues.

> "Within a class, function A calls function B and function B calls function C, where functions A, B, and C are distinctive."

Using RSF, this rule can be formally specified as:

```
((calls, Class1:Func1, Class2:Func2) AND
(calls, Class2:Func2, Class3:Func3) AND
(not_equal, Class1:Func1, Class2:Func2) AND
(not_equal, Class1:Func1, Class3:Func3) AND
(equal, Class1, Class2) AND
(equal, Class1, Class3))
```

If an abstract pattern rule can not be mapped to a concrete pattern rule (e.g., the pattern is defined by complex dynamic attributes), one needs to assess whether it is a *necessary* rule for the pattern recognition task in hand. A *necessary* abstract pattern rule specifies a distinct characteristic of the target pattern. A potential pattern instance that is missing such a characteristic would be disqualified from being an actual pattern instance. Based on the assessment, one may decide to proceed to the next step of ARM if the missing abstract pattern rules are *not* necessary, or to terminate the recognition task if *any* necessary abstract pattern rule is missing in the concrete pattern rules.

The second step is to translate the instantiated pattern description into pattern *queries*, written for one of the query and/or analysis tools supported by Dali. If the concrete pattern rules describe specific types of components and connectors, then tools based on a relational algebra such as SQL are suitable because they provide efficient and accurate matching on specific components and relations (connectors). If, on the other hand, the concrete pattern rules do not specify types of components or connectors, but rather allow for a wide range of possible realizations for a pattern, then tools that support more generalized searching criteria, such as the SAAMTool/IAPR toolset, should be used. A user can use the SAAMTool to specify a pattern as a graph and use attributed subgraph isomorphism provided by IAPR to match patterns. For example, the *Mediator pattern* description can be translated into an SQL query as follows:

```
SELECT DISTINCT c1.tcaller,
 c1.tcallee as mediator, c2.tcallee
INTO TABLE med
FROM calls c1, calls c2
WHERE c1.tcallee = c2.tcaller AND
```

```
c1.tcaller <> c1.tcallee AND
c1.tcaller <> c2.tcallee AND
classname(c1.tcaller)=classname(c1.tcallee)
AND
classname(c1.tcaller)=classname(c2.tcallee);
```

Finally, a concrete pattern recognition *plan* must be developed to specify the "key" component of the pattern that should be recognized first and the order in which the subsequent components should be detected. The queries for a "key" component should *not* depend on detection of other pattern components. The mediator component in the *Mediator pattern*, for example, serves as the communication hub between colleague components and thus is the key to recognizing this pattern. If part of the target pattern is designed and implemented using other lower-level patterns, it is necessary to develop concrete pattern recognition plans for each pattern component and the compound pattern.

3.2 Extracting a source model

The second phase of ARM is to extract a source model that represents a system's source elements and the relations between them. The output of this phase is a source model that contains the information that is used for detecting *necessary* pattern rules. For example, Table 1 shows some of the relations that Dali currently extracts from C++ programs (Kazman & Carriere, 1999). The relations needed for detecting the necessary pattern rules of the Presentation-Abstraction-Control (PAC) pattern[2] (Buschmann, et al., 1996) in our case studies are denoted by *.

Table 1: Typical set of source relations extracted by Dali.

Relation	From	To
calls *	function	function
contains	file	function
defines	file	class
has_subclass *	class	class
has_friend	class	class
defines_fn *	class	function
has_member *	class	variable
defines_var *	function	variable
has_instance *	class	variable
defines_global *	file	variable
var_access *	function	variable

[2] The PAC pattern is described in detail in section 4.1.

A complication is that patterns are revealed at different levels of abstraction (e.g., the function vs. the class level), thus different parts of the recognition plan may need to be applied to a source model at different levels of abstraction. Using abstraction techniques, such as the *aggregation* technique provided by Dali (Kazman & Carriere, 1999), lower level source model elements can be grouped into a higher level element without loss of information. Thus one can use it to bring the source model to appropriate levels of abstraction for pattern detection and architecture analysis.

3.3 Detecting and evaluating pattern instances

Detecting pattern instances using Dali is an automatic process in which one uses query tools to execute a recognition plan with respect to a source model. After running the recognition plan on the source model using the query tools, the detection output consists of all the pattern instance candidates. Human evaluation of these candidates is required to compare them with the designed pattern instances and determine which candidates are intended, which are false positives and false negatives. A false positive is a candidate which is not designed as a pattern instance, but is "detected" falsely as an instance. A false negative is a candidate which is designed as an instance, but is not detected as one.

One can try to improve the results (i.e., remove false positives and negatives) by modifying either the recognition plan or the source model and reiterating through ARM method. To improve the pattern recognition plan, one may choose another component of the pattern as the anchor and reorder the queries to form a new plan, or refine the query constraints for some of the pattern elements. If the source model extraction caused the deficiencies, an analyst needs to try to improve the extraction process by refining the existing extraction tools to catch the defects and/or incorporating other extraction tools to enhance the accuracy of source model, as described in (Kazman & Carriere, 1998).

However, if the source code is incomplete or if the pattern is defined by complex dynamic attributes, it may be impossible for the recognition technique to precisely detect all pattern instances. The evaluation process ends when Dali can detect the maximal set of true pattern instances, and the human analyst can explain the presence of false positive and the absence of false negative instances. The output is the set of validated pattern instances.

3.4 Reconstructing and analyzing the architecture

In the final step, the analyst uses a visualization tool, such as Rigi, to align the recognized architectural pattern instances with the designed pattern instances, organizing the other elements in the source model around the detected instances. The resultant architecture can be analyzed for deviations from the designed architecture.

4. CASE STUDIES

In an attempt to evaluate the applicability and generality of ARM, we applied it to two case studies where the systems were designed and developed with specific architectural patterns in mind. We obtained both source code and design documents for the applications from Informatique et Mathematiques Appliquees de Grenoble (IMAG) Institute in France.

4.1 SupraAnalyse system

The first application is a 25 KLOC system written in C++, called SupraAnalyse, that analyzes experimental data about human subjects' behavior when performing tasks using an interactive system (Lischetti & Coutaz, 1994). SupraAnalyse uses the Presentation-Abstraction-Control (PAC) pattern in its architectural design and implementation. The PAC pattern (Figure 4) defines a structure for interactive software systems in the form of a hierarchy of co-operating agents. Every agent is responsible for a specific aspect of the application's functionality and consists of three components: presentation, abstraction, and control. The *Presentation* component provides the visible interface; the *Abstraction* component maintains and accesses the data model; and the *Control* component manages intra-agent communications between the Presentation and Abstraction components and inter-agent communications with other PAC agents.

Based on the instantiation of PAC patterns in SupraAnalyse, we first developed a recognition plan which consists of a sequence of SQL queries. Because the internal structure of a PAC agent is designed using the Mediator pattern, we iterated the recognition plan development phase to fully specify a sub-plan for recognizing the Mediator pattern. Several extraction tools, including LSME (Murphy & Notkin, 1996), Imagix (Imagix, -) and SNiFF+ (SNiFF+, -), were used to extract a source model that was stored in an SQL database.

Before applying the recognition plan, the source model was simplified to function level and class level abstractions using the aggregation technique. That is, class information such as methods and member variables, was aggregated with class definition; and function information, such as local variable usage, was aggregated with function definitions. PAC pattern components were then detected at function level abstraction, and PAC agents were recognized at the class level.

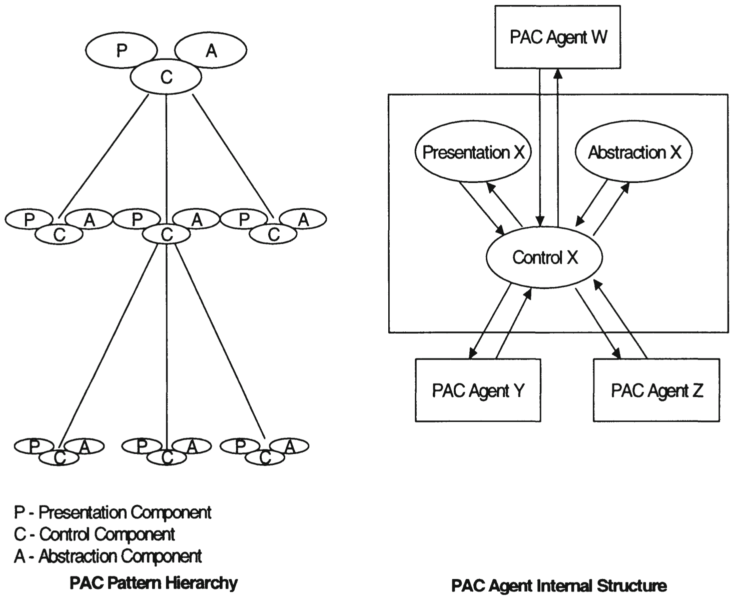


Figure 4: PAC patterns

Evaluation of the detection results was performed to identify false positives and false negatives. For example, the designed PAC agent “Ciment” is identified as a false negative because it can not be aligned to any detected pattern instance candidates. Subsequent iterations of ARM were taken to improve the source model extraction and recognition plan. We ended the iteration process when all false positives and false negatives were removed or explained by valid causes (such as incompleteness of the source code for the case where some class implementations were missing). For the false positive “Ciment” agent, further study of the source code shows that this designed agent is not implemented.

Finally, we re-constructed the as-implemented architecture (Figure 5) by aligning detected PAC agents with the intended PAC agents in the designed architecture, and grouping the unmatched detected agents together (at the bottom of Figure 5). Architecture conformance was analyzed to identify deviations of the as-implemented architecture from the documented architecture.

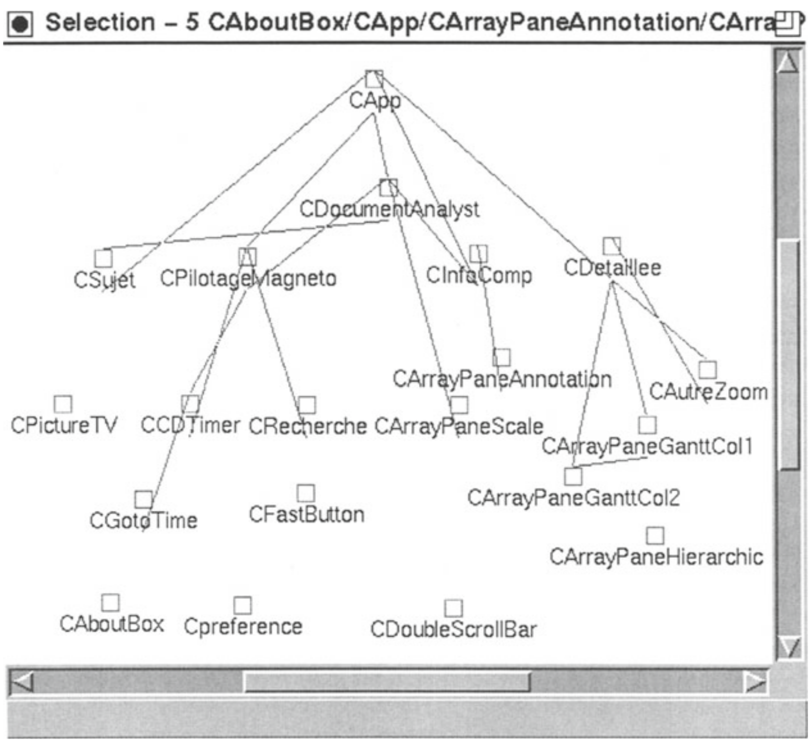


Figure 5: As-implemented architecture of SupraAnalyse using PAC patterns

The as-implemented architecture shows that there are relations that bridge layers of objects and thus violate the design principles of the PAC pattern. For example, agents “CSujet” and “CDetaillee” communicate directly with the top agent “CApp” and thus bridge over the “CDocumentAnalyst” agent. A further investigation of the layer bridging in the SupraAnalyse system was performed via searching for the *Layer-Bridging* pattern in the PAC agent hierarchy. ARM was applied again for this task. Because a layer may contain *any* type of component and because layer bridging can happen in several types of relations, an SQL pattern recognition plan was deemed inappropriate, since it would have involved listing all possible combinations of component and relation types. Instead we used SAAMTool to construct the Layer-Bridging pattern query as a graph. Nodes in the graph represent *any* type of component and edges represent *any* type of connector. The IAPR tool was then used to process the graphical query on a source model graph—the query posed as a subgraph isomorphism problem (Kazman & Burth, 1998). Three instances of Layer-Bridging pattern were detected. These instances represent problematic areas where the implementation of SupraAnalyse has drifted from the design, when we asked the authors of the system about the layer bridging, they said they were unaware of the presence of the design violations.

4.2 MATIS system

The second case study was conducted on a larger system (77 KLOC) called Multimodal Airline Travel Information System (MATIS): an interactive system which allows the end-user to obtain information about flight schedules using speech, mouse, keyboard, or a combination of these interfaces (Nigay & Coutaz, 1991), (Nigay & Coutaz, 1993). It was implemented in Objective C using the NeXTSTEP Application Development Kit.

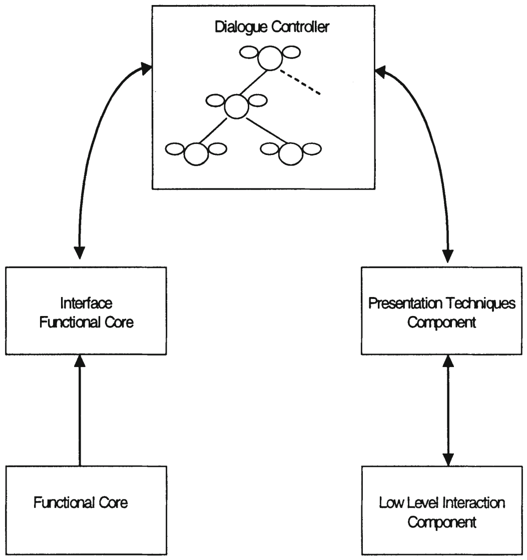

Dialogue Controller
Interface Functional Core
Presentation Techniques Component
Functional Core
Low Level Interaction Component

Figure 6: PAC-Amodeus pattern

The primary architectural pattern, the PAC-Amodeus model (see Figure 6) consists of 5 components organized symmetrically around a key component: the *Dialogue Controller (DC)*, which itself is designed using the PAC pattern. The *Functional Core (FC)* maintains domain data and performs domain-related functions. The *Interface with the Functional Core (IFC)* defines a set of interface objects to the Dialogue Controller and maps these interface objects into the formalism of the Functional Core.

The *Low Level Interaction Component (LLIC)* contains the toolkits that implement the physical interface between the user and the application. The *Presentation Techniques Component (PTC)* is a mediator between the Dialogue Controller and the Low Level Interaction Component, and controls the perceivable behavior of the application via input and output commands. The key component *Dialogue Controller* is responsible for task level sequencing, by creating a thread for each request received from PTC and linking the appropriate IFC objects to perform the request. The IFC and PTC components are abstraction layers to enhance portability.

Realizing that the DC component is the easiest to recognize as a PAC pattern instance, we formed our recognition plan as follows: first detect the PAC pattern instances and use these to identify the DC; second detect other components and hence the entire PAC-Amodeus pattern using the DC as the "anchor" of the pattern. The PAC pattern queries developed for SupraAnalyse were reused because they were applied to the elements and relations stored in the source model repository and therefore were not dependent on the particular language of implementation. Since the source code contains Objective C files and C++ files, language-specific extractors were developed and used to extract information from the system. A source model was created by combining the extraction results.

Running the PAC pattern queries, we detected 8 PAC agents. Evaluating these PAC agents against the design document shows that the DC is composed of 4 PAC agents; another 4 recognized PAC agents belong to other PAC-Amodeus components. The detected PAC agent information was then added to the repository to enrich the source model. Using the DC as the starting point, other PAC-Amodeus components were subsequently detected by executing the rest of the recognition plan.

After evaluating the detection results, we reconstructed the implemented architecture of MATIS, shown in Figure 7, using the recognized PAC-Amodeus instance. The PAC-Amodeus structure is evident, but there are several anomalies that need to be investigated. For example, the implemented architecture shows that the FC component, which was designed as an SQL database to process requests sent from the IFC, is missing. Investigation of the source code confirms that these requests are handled by a function in IFC that *simulates* the database processing by returning pre-defined values to certain requests.

As another example, Figure 7 shows that the LLIC calls the DC directly, bridging over the PTC. This clearly violates the design of the PTC component as a layer between the DC and LLIC.

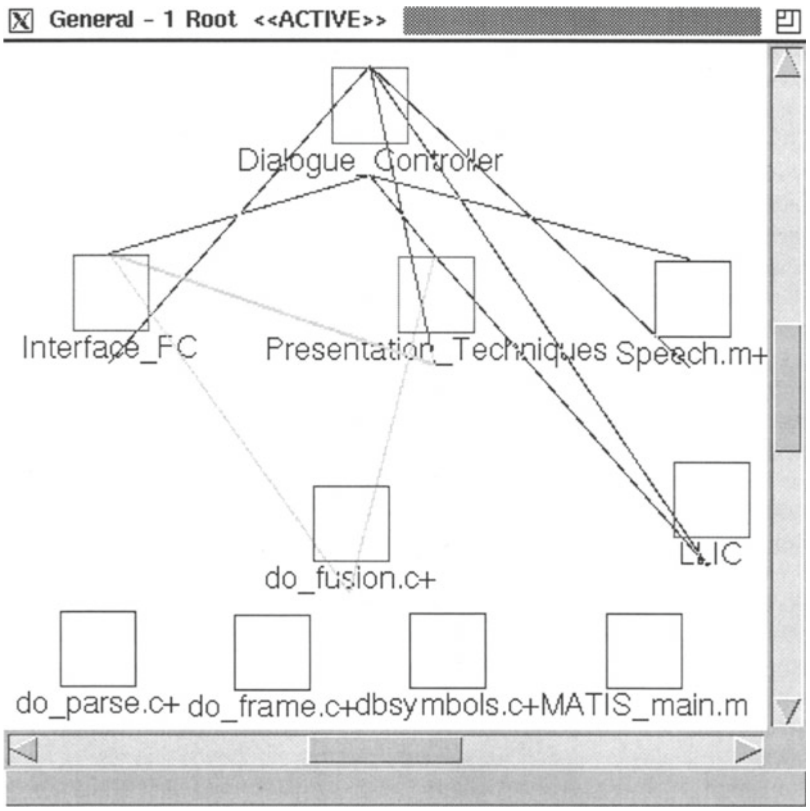


Figure 7: Recognized PAC-Amodeus pattern in MATIS. Solid lines represent calls relations and shaded lines represent variable access relations

5. LESSONS LEARNED

These case studies both used *patterns* as the primary technique for reconstructing software architectures. They demonstrate the usefulness of ARM in assessing, planning and executing pattern recognition tasks. A recognition plan can be laid out to recognize a pattern by a) recognizing nested lower-level patterns first; b) recognizing the pattern's key element; and c) recognizing other elements of the pattern and hence the entire pattern. The pattern matching process is facilitated by using automated query and analysis tools. If an iteration of ARM can not be completed because the exit conditions for a step can not be met, proper assessment of the task should be conducted to identify the causes of detection deficiencies and to provide guidelines for future efforts to improve the pattern detection.

This process is efficient both in terms of the analyst's time and in terms of the amount of processing required to do pattern recognition. Consider, for example, the tools presented here: SQL queries to match patterns are quite efficient (as long as appropriate database indices have been built in advance on the tables of interest), and IAPR pattern-matching, while in principle NP-hard, can be rendered tractable by the judicious use of features that limit the search space, as reported in (Kazman & Burth, 1998).

The time spent in learning and using ARM can be amortized over several architecture reconstruction tasks performed on similar systems (written in the same language and/or using the same design patterns). Queries developed from previous applications of ARM may be reused in executing one or more pattern recognition tasks, as we showed by reusing the PAC pattern queries.

The case studies also provide evidence that static analysis of source code is *not* always sufficient for pattern recognition. Patterns that are implemented using only static mechanisms can be recognized from a source model containing static source artifacts. Patterns whose implementation involves dynamic mechanisms will require extraction of dynamic information, such as process spawning, instances of interprocess communication (IPC), and run-time procedure invocation. In the MATIS implementation, for example, object variables are dynamically typed. That is, an object variable is declared to be a generic type, and assigned specific class types at run time. The best way to solve this problem is to extract the object-type information at run time. However, due to the lack of access to the NeXT Application Development Kit environment (including its class libraries), we could not execute the system or use dynamic analysis tools to extract the missing object-type information. Fortunately, the object variables were never assigned to more than one type in MATIS. Therefore, we were able to use the static object creation and assignment information to resolve the type of each object. This suggests that extracting dynamic information of a system at run time will sometimes be necessary even in reconstructing a static architecture.

6. CONCLUSIONS

Using design patterns in software design has become a widely used technique for achieving a high quality architecture. Reconstructing architectures of systems that were designed and developed with design patterns has traditionally been approached through manual source code inspections (Schull, et al., 1996). In this paper, we presented ARM—a semi-automatic analysis method—to reconstruct architectures based on recognized instances of design patterns. ARM is an iterative and interpretive process; a human is an integral part of the loop, to evaluate the results and determine what patterns to apply in the subsequent iteration. Two aspects differentiate ARM from other approaches for pattern recognition. One, ARM clearly distinguishes abstract pattern description from concrete pattern instantiation and uses the latter to guide pattern detection. Two, using automated tools to perform pattern matching makes the pattern recognition process less error-prone, compared to manual inspections. Upon the reconstruction of the system's architecture, we can analyze conformance of the software to the documented design patterns.

To further validate the usefulness and applicability of ARM, more case studies need to be conducted on systems in various application domains. Another area of future work is to incorporate approximate pattern matching techniques into ARM. The associated metrics to measure the dissimilarity between the pattern query and the actual pattern instance need to be further studied and established.

Finally, to make ARM still more cost-effective, a pattern knowledge base could be built to provide recognition plans tailored for common instantiations of a given pattern.

REFERENCES

Bass, L., Clements, P., Kazman, R. (1998), *Software Architecture in Practice*, Addison-Wesley.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996), *Pattern-Oriented Software Architecture*, Wiley.

Chen, Y., Nishimoto, M., Ramamoorthy, C. (1990), The C Information Abstraction System, *IEEE Transactions on Software Engineering*, 3, 325-334.

Finnigan, P., Holt, R. C., *et al.* (1997), The Software Bookshelf, *IBM Systems Journal,* 36(4), 564-593.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994), *Design Patterns*, Addison Wesley.

Imagix Corporation, http://www.imagix.com

Portable Bookshelf, http://turing.toronto.edu/~holt/pbs

Johnson, J. (1993), Identifying Redundancy in Source Code Using Fingerprints, *Proceedings of CASCON '93,* 171-183.

Kazman, R., Abowd, G., Bass, L., Webb, M. (1994), SAAM: A Method for Analyzing the Properties of Software Architectures, *Proceedings of the 16th International Conference on Software Engineering*, 81-90, IEEE Computer Society Press.

Kazman, R. (1996), Tool Support for Architecture Analysis and Design, *Joint Proceedings of the SIGSOFT '96 Workshops (ISAW-2)*, 94-97, ACM.

Kazman, R., Burth, M. (1998), Assessing Architectural Complexity, *Proceedings of 2nd Euromicro Working Conference on Software Maintenance And Reengineering (CSMR)*, 104-112, IEEE Computer Society Press.

Kazman, R., Carrière, S. J. (1998), View Extraction and View Fusion in Architectural Understanding, *Fifth International Conference on Software Reuse*, 290-299.

Kazman, R., Carrière, S. J. (1999), Playing Detective: Reconstructing Software Architecture from Available Evidence, *Automated Software Engineering,* 6:2, April 1999, to appear.

Kontogiannis, K., DeMori, R., Bernstein, M., Merlo, E. (1994), Localization of Design Concepts in Legacy Systems, *Proceedings of International Conference on Software maintenance '94*, 414-423.

Lischetti, N., Coutaz, J. (1994), *Supraanalyse de supratel.* Technical report, Informatique et Mathematiques Appliquees de Grenoble (IMAG).

Murphy, G., Notkin, D. (1996), Lightweight Lexical Source Model Extraction, *ACM Transactions on Software Engineering and Methodology*, 5(3), 262-292.

Murphy, G., Notkin, D., Sullivan, K. (1995), Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 18-28, ACM Press.

Nigay, L., Coutaz, J. (1993), A Design Space for Multimodal Systems: Concurrent Processing and Data Fusion, *Proceedings of InterCHI '93*, ACM Press.

Nigay, L., Coutaz, J. (1993), Building User Interfaces: Organizing Software Agents, *Proceedings of ESPRIT '91*, 707-719.

Reasoning Inc., http://www.reasoning.com

SNiFF+, http://www.seed.arch.adelaide.edu.au/docs/sniff_online

Schull, F., Melo, W., Basili, V. (1996), *An Inductive Method for Discovering Design Patterns from Object-Oriented Software Systems*, UMIACS-TR-96-10, University of Maryland.

Storey, M., Muller, H., Wong, K. (1996) Manipulating and Documenting Software Structures, *Software Visualization,* World Scientific.

UIMS Tool Developers Workshop (1992), A Metamodel for the Runtime Architecture of an Interactive System, *SIGCHI Bulletin*, 24(1), 32-37.

Wong, K., Tilley, S., Muller, H., Storey, M. (1994), Programmable Reverse Engineering, *International Journal of Software Engineering and Knowledge Engineering*, 4(4), 501-520.

Woods, S. G., Yang, Q. (1995), Program Understanding as Constraint Satisfaction, *Proceedings of the IEEE Seventh International Workshop on Computer-Aided Software Engineering (CASE-95)*, IEEE Computer Society Press.

Yeh, A., Harris, D., Chase, M. (1997), Manipulating Recovered Software Architecture Views, Proceedings of ICSE 19, 184-194, ACM Press.

Behaviour Analysis of Software Architectures

Jeff Magee, Jeff Kramer, and Dimitra Giannakopoulou
Department of Computing, Imperial College of Science, Technology and Medicine,
180 Queensgate, London, SW7 2BZ, U.K.
{jnm,dg1,jk}@doc.ic.ac.uk

Key words: Software architecture, behaviour analysis

Abstract: The overall structure of a system described by a set of components and their interconnections is termed its software architecture. In this paper, we associate behavioural specifications with components and use these specifications to analyze the overall system architecture . The approach is based on the use of Labelled Transition Systems to specify behaviour and Compositional Reachability Analysis to check composite system models. The architecture description of a system is used directly in the construction of the model used for analysis. Analysis allows a designer to check whether an architecture satisfies the properties required of it. The paper uses examples to illustrate the approach and discusses some open questions arising from the work.

1. INTRODUCTION

Software architecture has been identified as a promising approach to bridging the gap between requirements and implementations in the design of complex systems. Software architecture describes the gross organisation of a system in terms of its components and their interactions. The initial emphasis in Software architecture specification has thus been in capturing system structure [5,8,13]. The authors have previously published papers on the use of the architecture description language Darwin for specifying the structure of distributed systems and subsequently directing the construction of those systems [8,9,10]. Darwin can also be used to organise CORBA based distributed systems [11]. Darwin describes a system in terms of components, which manage the implementation of services. Interconnection structure is specified by bindings between the services required and provided by component instances. Darwin has both a graphical and a textual form with appropriate tool support [9,12].

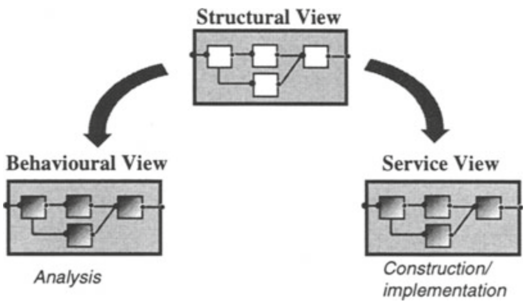


Figure 1. Common structural view with service and behavioural views

In this paper, we describe the use of Darwin structural descriptions as a framework for behaviour analysis rather than system construction. Darwin has been designed to be sufficiently abstract to support multiple views (cf. [7]), two of which are the behavioural view (for behaviour analysis) and the service view (for construction) (Figure 1). Each view is an elaboration of the basic structural view: the skeleton upon which we hang the flesh of behaviour specification or service implementation [14].

In previous papers, we have discussed the use of Darwin to produce the service view, with components providing and requiring services at their interfaces and with implementation definitions for the primitive components. For example, when used to structure CORBA systems [11], the computational behaviour of Darwin primitive components is determined by CORBA object implementations and these object implementations interact via interfaces specified in IDL using the ORB in the usual way. Primitive components encapsulate objects and specify their instantiation, their required interfaces and provided interfaces. As depicted in figure 2, a primitive component may embed one or more objects.

In this paper we concentrate on the behavioural view using Labelled Transition Systems (LTS) for behaviour specification and analysis. The analysis approach is Compositional Reachability Analysis CRA [4]. We have developed techniques for analysing system models in the CRA setting with respect to both safety [2] and liveness [3] properties. The techniques are supported by software tools, which provide for automatic composition, analysis, minimisation, animation and graphical display. We first describe the relationship between components and their behavioural specifications.

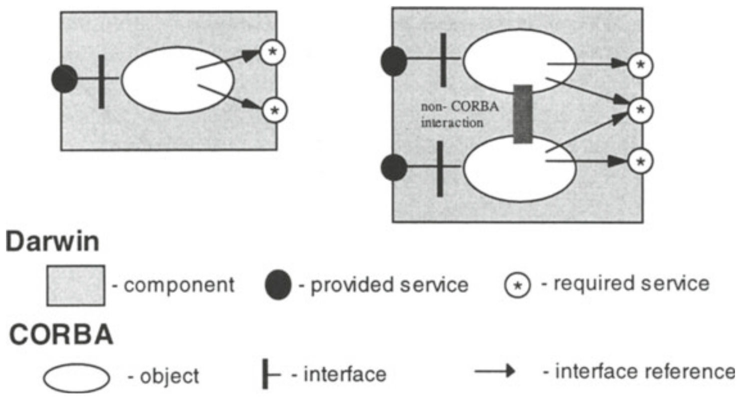


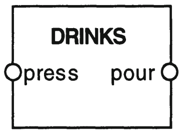
Figure 2. Embedding objects in components

2. PRIMITIVE COMPONENTS

A primitive component is one with no substructure of components. In the service view of architecture, a primitive component has an implementation defined by an object or objects programmed in a programming language such as C++. In the behavioural view, a primitive component is defined as a finite state LTS. The example of figure 3 depicts the Darwin graphical and textual description of a primitive component with two interfaces.

In the behavioural view, we do not distinguish between provided and required services, service access points are simply declared as *portals*. Consequently, implementation details such as invocation direction can be deferred, although, in many cases, it is obvious from the behavioural model as to which component is providing a service and which is using it.

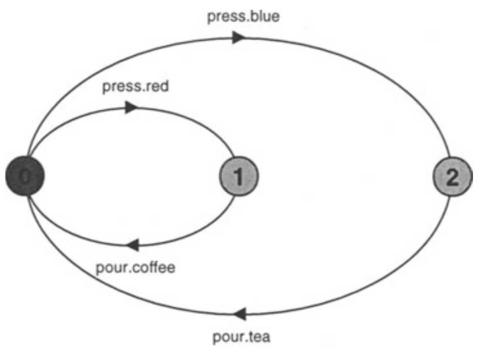
A major objective of our work in architectural analysis is to provide tools that are both accessible and usable by practising engineers. To this end, we originally conceived that the behaviour of primitive components should be specified graphically as state transition diagrams since these should be familiar in one form or another to software engineers. However, it quickly became apparent that this is an extremely cumbersome method for other than trivial behaviour specifications. With our focus on actions rather than states in specifying behaviour, it was natural to use process algebra as a concise notation for describing behaviour. However, it is unlikely that most software engineers have a working knowledge of process algebra. To mitigate this problem, we have included the facility to depict textual specifications as labelled transition diagrams. These diagrams may be animated, by an interactive behaviour simulation, to check that the specification corresponds to the engineer's intuition.



```
interface BUTTON  {red; blue;}

interface BEVERAGE{coffee; tea;}

component DRINKS {
  portal press:BUTTON;
  portal pour :BEVERAGE;
}
```

Figure 3. Darwin description of DRINKS component

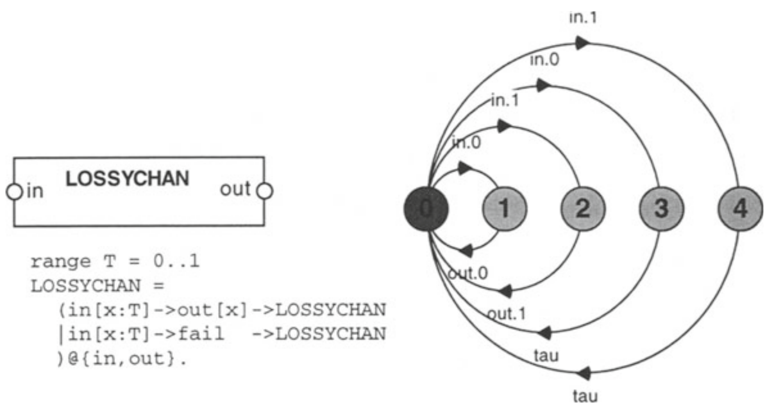
The behaviour of the drinks component is modelled in Figure 4 both graphically as a Labelled Transition System and textually in our process algebra notation FSP (Finite State Processes).



```
DRINKS = (press.red  -> pour.coffee -> DRINKS
         |press.blue -> pour.tea    -> DRINKS
         ) @ { press, pour}.
```

Figure 4. Behavioural description of DRINKS component

Primitive components are defined as finite state processes in FSP using action prefix "->" and choice "|". If `x` is an action and `P` a process then `(x->P)` describes a process that initially engages in the action `x` and then behaves exactly as described by `P`. If `x` and `y` are actions then `(x->P|y->Q)` describes a process which initially engages in either of the actions `x` or `y`. After the first action has occurred, the subsequent behaviour is described by `P` if the first action was `x` and `Q` if the first action was `y`. Thus the `DRINKS` component offers a choice of the actions `press.red` and `press.blue`. As a result of engaging in one of these actions the appropriate drink is poured. The behavioural view does not distinguish between input and output actions although, as in the example, input actions generally form part of a choice offered by a component while output actions do not. The `@{press,pour}` states that all actions labelled or prefixed by `press` or `pour` can be shared with other components. The next example is a component that has internal actions that cannot be shared with other components. Figure 5 gives the Darwin graphical description for the primitive component `LOSSYCHAN` together with its behaviour modelled in FSP and the corresponding LTS diagram.



```
range T = 0..1
LOSSYCHAN =
   (in[x:T]->out[x]->LOSSYCHAN
   |in[x:T]->fail   ->LOSSYCHAN
   )@{in,out}.
```

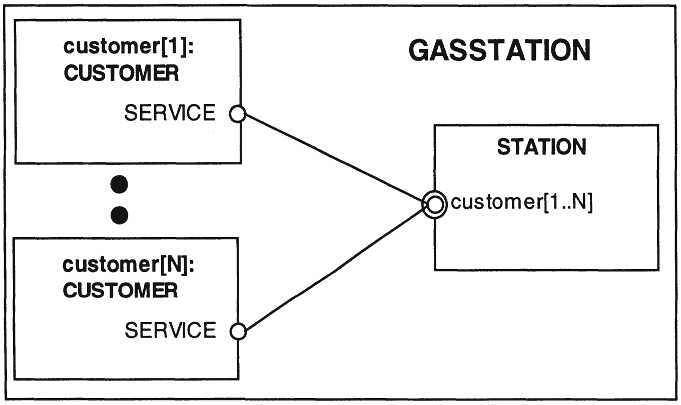
Figure 5. LOSSYCHAN component

The component `LOSSYCHAN` models a channel which inputs values in the range `0..1` and then either outputs the value or fails. In other words, the component models a transmission channel that can lose messages. Failure is modelled by non-deterministic choice on the input, which leads to the internal action `fail`, if failure is chosen. Since `fail` does not appear at the interface of the component, it becomes the silent action `tau` in the LTS diagram for the component. In many Architectural Description Languages, `LOSSYCHAN` would be represented as a connector rather than a component [1,13]. However, Darwin does not have a separate connector construct. Connectors can be distinguished as a particular class of components. It is clear from the above that connectors are modelled in exactly the same way as components.

The modelling notation FSP—finite state processes—includes guarded choice, local processes and conditional processes. However, these are syntactic conveniences to allow concise model definition. Definitions using these constructs can all be expressed using action prefix, choice and recursion as described in this section.

3. COMPOSITE COMPONENTS

A composite component is constructed from interconnecting instances of more primitive components. A composite component defines a structure and no additional behaviour. Its behaviour can therefore be computed based on this structure and the behaviour of its components.



```
const int N = 3; //#customers

interface SERVICE {
  prepay(int); gas(int);
}
component CUSTOMER {
  portal
    SERVICE;
}
component STATION {
  portal
    customer[1..N]:SERVICE;
}
```

```
component GASSTATION {
  inst
    STATION;
  forall i = 1 to N {
    inst
      customer[i]:CUSTOMER;
    bind
      customer[i].SERVICE
       --STATION.customer[i];
  }
}
```

Figure 6. GASSTATION composite component

To illustrate composition, we will use the gas station problem, originally stated in [16] and more recently addressed in [2,17]. The gas station problem concerns a set of *N* customers who obtain gas by prepaying a cashier who activates one of *M* pumps to serve the customer. The overall `GASSTATION` component is depicted in figure 6.

In an implementation such as CORBA discussed in the introduction, Darwin bindings (drawn as arcs between portals) are generally references to objects. In the behavioural view, a binding denotes an action shared between two components. Each customer in figure 6 shares the actions `prepay` and `gas`, which constitute the `SERVICE` interface, with the `STATION` component. Component instances in the behavioural view are finite state processes as described in the previous section. The composite behaviour is the parallel composition of these processes. Consequently, the behaviour of `GASSTATION` is the parallel composition of its constituent components:

```
||GASSTATION = (customer[1..N]:CUSTOMER || STATION).
```

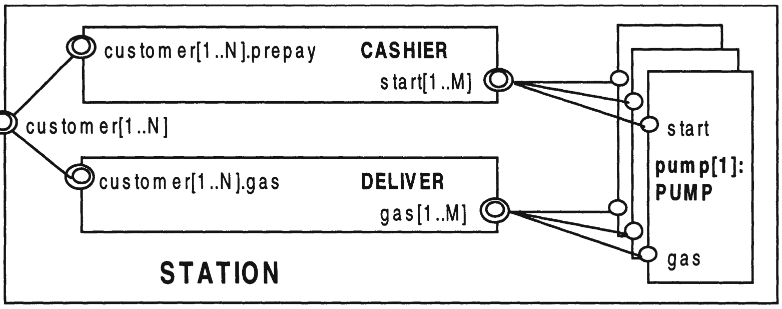
Note that to create multiple copies of `CUSTOMER` we use process labelling. Each action label of the customer process (namely `prepay` and `gas`) is prefixed with the process label. Thus customer 1 has the action labels `customer[1].prepay` and `customer[1].gas`. The `STATION` is itself a composite component consisting of the cashier and one or more pumps as depicted in figure 7. A `DELIVER` component is also required to associate pump actions with customer actions. The need for this component is discussed later in the paper.

A binding in Darwin always denotes a shared action in the behavioural view. Shared actions are the means by which processes synchronise and interact in FSP. It is sometimes necessary to relabel actions to ensure that the shared action has the same name in all the processes that share that action. Re-labelling is required in the FSP description of the `STATION` component based on the particular bindings:

```
||STATION = (CASHIER || pump[1..M]:PUMP || DELIVER)
            /{pump[i:1..M].start/start[i],
              pump[i:1..M].gas/gas[i]}
             @{customer}.
```

The general form of the relabeling function is:
/{*newlabel*_1/*oldlabel*_1,... *newlabel*_n/*oldlabel*_n}.

This section has outlined how the FSP composition expressions for the behavioural model can be generated directly from the Darwin composite component structure. In the next section, we discuss analysis using the behavioural model.



```
const M = 2; //#pumps
component STATION {
  portal customer[1..N]:SERVICE;
  inst CASHIER;
  inst DELIVER;
  forall i = 1 to N bind
     customer[i].prepay -- CASHIER.customer[i].prepay;
     customer[i].gas    -- DELIVER.customer[i].gas;
  forall i = 1 to M {
    inst pump[i]:PUMP;
     bind
       pump[i].start -- CASHIER.start[i];
       pump[i].gas   -- DELIVER.gas[i];
  }
}
```

Figure 7. STATION composite component

4. ANALYSIS

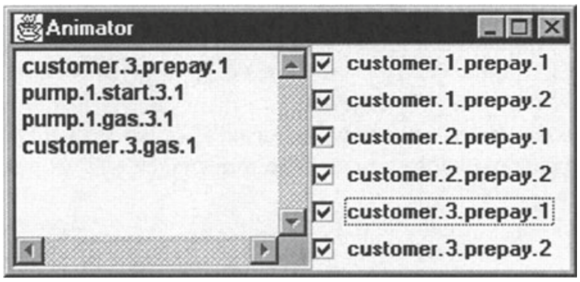
The complete behavioural model for the gas station is listed in figure 8. It includes behaviour definitions for the primitive components, CUSTOMER, CASHIER, PUMP and DELIVER. A CUSTOMER makes a prepayment of some amount (a) chosen from the range (A) and then inputs some amount of gas (x). The process definition includes a test to check that the amount of gas actually delivered is the same as the amount paid for. In this simplified model of the gas station, the cashier does not give change and pumps are expected to deliver the amount of gas that has been paid for. The CASHIER starts any pump that is ready and passes to it the identity of the customer (c) and the amount of gas required (x). The PUMP outputs the correct amount of gas, which is delivered to the CUSTOMER by the DELIVER component. The composition expressions for the composite components `STATION` and `GASSTATION` are as described in the previous section.

```
const N = 3     //number of customers
const M = 2     //number of pumps
range C = 1..N  //customer range
range P = 1..M  //pump range
range A = 1..2  //amount of money or Gas

CUSTOMER = (prepay[a:A]->gas[x:A]->
               if (x==a) then CUSTOMER else ERROR).
CASHIER =
  (customer[c:C].prepay[x:A]->start[P][c][x]->CASHIER).
PUMP =
  (start[c:C][x:A] -> gas[c][x] -> PUMP).
DELIVER=
  (gas[P][c:C][x:A] -> customer[C].gas[x] -> DELIVER).

||STATION = (CASHIER || pump[1..M]:PUMP || DELIVER)
            /{pump[i:1..M].start/start[i],
              pump[i:1..M].gas/gas[i]} @{customer}.

||GASSTATION =  (customer[1..N]:CUSTOMER ||STATION).
```

Figure 8. Gas station behavioural model

Animation

Our analysis tool *LTSA* (labelled transition system analyser) allows a user to explore different execution scenarios using the behavioural model.



Animator

```
customer.3.prepay.1
pump.1.start.3.1
pump.1.gas.3.1
customer.3.gas.1
```

- ✓ customer.1.prepay.1
- ✓ customer.1.prepay.2
- ✓ customer.2.prepay.1
- ✓ customer.2.prepay.2
- ✓ customer.3.prepay.1
- ✓ customer.3.prepay.2

Figure 9. Animating the gas station

To do this, the user must specify the set of actions that he/she wants to control. The controlled set is defined by a menu, which for figure 9 is:

```
menu RUN = {customer[C].prepay[A]}
```

Figure 9 depicts the trace of actions that result from instigating a prepay action from customer 3. The cashier allocates pump 1, which delivers the requisite gas to the customer via the DELIVER process.

Reachability Analysis

Animation allows a user to explore different execution scenarios, however, it does not allow general properties concerning the model to be checked. For example, does a customer *always* receive the correct amount of gas? Reachability analysis performs an exhaustive search of the state space to detect ERROR and deadlock states (no outgoing transitions). In fact the behaviour model of figure 7 has a bug that permits incorrect behaviour. The output of the analyser is shown below:

```
property customer.3:CUSTOMER violation.
property customer.2:CUSTOMER violation.
property customer.1:CUSTOMER violation....
States Composed: 3409 Transitions: 11862 in 1468ms
Trace to property violation in customer.2:CUSTOMER:
customer.1.prepay.1
pump.1.start.1.1
customer.2.prepay.2
pump.1.gas.1.1
customer.2.gas.1
```

The output shows that a property violation in each of the customer components is detected. In addition, an example trace, which causes one of the violations, is produced. Remembering that the CUSTOMER model requires that the amount of gas delivered to the customer should be the amount paid for, the trace is an execution in which customer 2 gets the gas paid for by customer 1. This error is essentially the same as the race condition discussed in [17]. The error in the model is that the DELIVER process delivers gas to *any* ready customer C rather than to the customer identity c passed to it by the cashier. The corrected DELIVER process is:

```
DELIVER
   =(gas[P][c:C][x:A] -> customer[c].gas[x] -> DELIVER).
```

Safety properties

We can specify safety properties that a composition of components must satisfy using property automata [2]. These specify the set of all traces that satisfy the property for a particular action alphabet. If the model can produce traces, which are not accepted by the property automata, then a violation is detected during reachability analysis. For example, the following automaton specifies that for, two customers, if one customer makes a payment then he or she should get gas before the next customer makes a payment. In other words, service should be FIFO.

```
range T = 1..2
property
   FIFO      = (customer[i:T].prepay[A] -> PAID[i]),
   PAID[i:T] = (customer[i].gas[A]      -> FIFO
               |customer[j:T].prepay[A] -> PAID[i][j]
               ),
   PAID[i:T][j:T] = (customer[i].gas[A] -> PAID[j]).
```

A gas station with a single pump satisfies this property, however, a station with two pumps does not and leads to the following violation:

```
Composing
 property FIFO violation.
States Composed: 617 Transitions: 1398 in 94ms
Trace to property violation in FIFO:
customer.1.prepay.1
pump.1.start.1.1
customer.2.prepay.1
pump.2.start.2.1
pump.2.gas.2.1
customer.2.gas.1
```

The trace describes the scenario in which customer 1 pays first and gets pump 1 followed by customer 2 paying and getting pump 2. Clearly in a two pump system, pump 2 can finish first, thereby violating the FIFO property.

Liveness properties

The LTSA analysis tool allows behavioural models to be checked against specific liveness properties specified in Linear Temporal Logic. However, we have found a check for a general liveness property which we term *progress* to provide sufficient information on liveness in many examples. Progress asserts that in an infinite execution of the system being modelled, all actions can occur infinitely often. In the gas station example, it would assert that customers will always eventually be served. In performing the progress check, we assume fair choice which means that if an action is eligible infinitely often, then it is executed infinitely often. With this assumption, the progress check finds no problem with the gas station. However, we can examine the behaviour of the system under different scheduling constraints by applying action priority. For example, the system below states that the actions of customer 1 have lower priority than other actions:

```
||GASSTATION = (customer[1..N]:CUSTOMER ||STATION)
                >>{customer[1]}.
```

Unsurprisingly, this causes a progress check violation since it is now possible for the cashier to ignore customer 1 in favour of other customers. Customer 1 may never be served. The tool gives the following output.

```
Progress violation for actions:
  {customer.1.prepay.1, customer.1.gas.1, customer.1.gas.2,
customer.1.prepay.2, pump.1.start.1.1, pump.2.start.1.1,
pump.1.start.1.2, pump.2.start.1.2, pump.1.gas.1.1,
pump.1.gas.1.2...........}
   Trace to terminal set of states:
   Actions in terminal set:
   {customer.2.prepay.1, customer.2.gas.1, customer.2.gas.2,
customer.2.prepay.2, customer.3.prepay.1, customer.3.gas.1,
customer.3.gas.2, customer.3.prepay.2, pump.1.start.2.1,
pump.2.start.2.1...........}
```

This includes the set of actions that do not occur infinitely often in the system and the set of action that can occur infinitely often. It is clear that actions for customer 1 occur in the former set and the actions for customer 2 in the latter. The tool gives a trace that leads to the execution in which the violation occurs. In the example, this trace is empty, as customer 1 never gets an opportunity to get gas.

5. DISCUSSION & CONCLUSIONS

We have presented an approach that associates behaviour descriptions with architectural components and supports behaviour analysis of the composition of these components according to the software architecture. Although relatively small, the example exhibits non-trivial behaviour. It demonstrates that we can produce concise and flexible behavioural models in which it is easy to add additional components and interactions. In the gas station, it is trivial to modify the numbers of customers and pumps. In fact, the gas station as presented is an instantiation of a common distributed software architecture style known as a multi-server or multithreaded server.

In a multi-server system, a separate server thread allocated by an administrator thread handles each client request.

In the introduction we stated that we could use the same structural description for system construction as for behaviour modelling. This is not always the case. For example, the gas station behavioural view includes the `DELIVER` component which routes pump actions to customers. This component would not appear in the service view since this routing would be implicit in the service invocation mechanism. `DELIVER` is modelling an aspect of architectural connection and it is specific to the behavioural view. In other words, we recognise that there is a need to augment the structural description with connector components for the behavioural view of architecture. In contrast to Wright [1] we have resisted requiring that a connector component is *always* interposed between application components since this seems to lead to large numbers of auxiliary actions.

An issue that always arises when considering exhaustive state space search methods is scalability. We have used the current toolset, which has not yet been optimised for performance, to analyse an Active Badge System[21] in which the final model has 566,820 reachable states and 2,428,488 possible transitions. This took 400 seconds to construct and check on a 200MHz Pentium Pro and required 170Mb of store. Although not addressed in this paper, our tools support compositional reachability analysis in which intermediate composite components can be minimised with respect to their interface actions using observational equivalence. Previous work [15] has addressed the problem of intermediate state explosion.

We believe that analysis and design are closely inter-linked activities which should proceed hand in hand. The FSP notation and its associated analysis tool *LTSA* have been carefully engineered to facilitate an incremental and interactive approach to the development of component based systems. Analysis and animation can be carried out at any level of the architecture. Consequently, component models can be designed and debugged before composing them into larger systems. The analysis results are easily related to the architectural model of interconnected components. The *LTSA* analysis tool described in this paper is written in Java™ and can be run as an application or applet. It is available at http://www-dse.doc.ic.ac.uk/~jnm. The approach we have described in this paper to analysing component-based systems is a general one that is not restricted to a particular tool-set. For example, CSP/FDR [6,19] has been used with the architectural description language Wright[1] and both LOTOS/CADP [18] and Promela/SPIN [20] have been used in the context of analysing software architectures. The objective, whatever the tool, is to use behaviour analysis during design to discover architectural problems early in the development cycle.

REFERENCES

[1] Allen R. and Garlan D., A Formal Basis for Architectural Connection, *ACM Transactions on Software Engineering Methodology* TOSEM, 6 (3), July 1997, 213-249.

[2] Cheung S.C. and Kramer J., Checking Subsystem Safety Properties in Compositional Reachability Analysis, *18th IEEE Int. Conf. on Software Engineering (ICSE-18)*, Berlin, 1996), 144-154.

[3] Cheung S.C., Giannakopoulou D., and Kramer J., Verification of Liveness Properties using Compositional Reachability Analysis, *6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 97)*, Zurich, Sept. 1997), LNCS 1301, (Springer-Verlag), 1997, 227-243.

[4] Giannakopoulou D., Kramer J. and Cheung S.C., Analysing the Behaviour of Distributed Systems using Tracta, *Journal of Automated Software Engineering*, special issue on Automated Analysis of Software (to appear), vol. 6(1). R. Cleaveland and D. Jackson, Eds.

[5] Garlan D. and Perry D.E., Introduction to the Special Issue on Software Architecture, *IEEE Transactions on Software Engineering*, 21 (4), April 1995, 269-274.

[6] Hoare, C.A.R., Communicating Sequential Processes, *Prentice-Hall*, Englewood Cliffs, N.J., 1985.

[7] Kruchten P.B., The 4+1 Model of Architecture, *IEEE Software*, 12 (6), Nov. 1995, 42-50.

[8] Magee J., Dulay N., Eisenbach S., Kramer J., Specifying Distributed Software Architectures, *5th European Software Engineering Conference (ESEC '95)*, Sitges, September 1995), LNCS 989, (Springer-Verlag), 1995, 137-153.

[9] Magee J., Dulay N. and Kramer J., Regis: A Constructive Development Environment for Distributed Programs, *Distributed Systems Engineering Journal*, 1 (5), Special Issue on Configurable Distributed Systems, (1994), 304-312.

[10] Magee J. and Kramer J., Dynamic Structure in Software Architectures, *4th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 4)*, San Francisco, October 1996), SEN, Vol.21, No.6, November 1996, 3-14.

[11] Magee J., Tseng A., Kramer J., Composing Distributed Objects in CORBA, *Third International Symposium on Autonomous Decentralized Systems (ISADS 97)*, Berlin, Germany, April 9 - 11, 1997.

[12] Ng K., Kramer J. and Magee J., Automated Support for the Design of Distributed Software Architectures, *Journal of Automated Software Engineering (JASE)*, 3 (3/4), Special Issue on CASE-95, (1996), 261-284.

[13] Shaw M., et al., Abstractions for Software Architecture and Tools to Support Them, *IEEE Transactions on Software Engineering*, 21 (4), April 1995, pp 314-335.

[14] Kramer J. and Magee J., Exposing the Skeleton in the Coordination Closet, *2nd IEEE International Conference on Coordination Models and Languages*, Coord '97, Berlin , September 1997), LNCS 1282, (Springer-Verlag), Sept 1997, pp. 18-31.

[15] Cheung S.C. and Kramer J., Context Constraints for Compositional Reachability Analysis, *ACM Transactions on Software Engineering Methodology TOSEM*, 5 (4), (1996), 334-377.

[16] Hembold, D. and Luckham, D.C., Debugging Ada Tasking Programs, *IEEE Software*, 2(2), March 1985, 47-57.

[17] Naumovich, G., Avrunin G.S., Clarke L.A. and Osterweil L.J. Applying Static Analysis to Software Architectures, *6th European Software Engineering Conference / 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 97)*, Zurich, Sept. 1997), LNCS 1301, (Springer-Verlag), 1997, 77-93.

[18] Jean-Pierre Krimm and Laurent Mounier. Compositional state space generation from LOTOS programs. In Ed Brinksma, editor, *Proceedings of TACAS'97 (Tools and Algorithms for the Construction and Analysis of Systems)*, Enschede, The Netherlands, April 1997. Springer Verlag.

[19] Formal Systems, Failues Divergence Refinement: FDR 2.0 User Manual, ed. *Formal Systems (Europe)*, Oxford, U.K, August 1996.

[20] Holtzman G.J., The Model Checker SPIN, *IEEE Transactions on Software Engineering,* 23(5) May 1997, 279-295.

[21] Magee J., Kramer J. and Giannakopoulou D., Analysing the Behaviour of Distributed Software Architectures: a Case Study, *5th IEEE Workshop on Future Trends in Distributed Computing Systems*, FTDCS'97, Tunsia, October 1997.

Rearchitecting Legacy Systems—Concepts and Case Study

Wolfgang Pree[1] & Kai Koskimies[2]
[1]University of Constance, Germany & [2]Nokia Research Center, Helsinki, Finland
pree@acm.org, kai.koskimies@research.nokia.com

Key words: Creation and evolution of architectures, product line architectures, framelets, automated configuration, dynamic architectures

Abstract: Legacy systems, no matter which architectural style they rely on, contain numerous pieces of source code with very similar functionality. We see these system aspects as a good starting point for rearchitecting legacy systems. The goal is the evolution of the legacy system architecture towards a product line architecture which incorporates the originally replicated system aspects as reusable, ideally self-configuring components. This paper presents the concepts which we regard as necessary and/or useful for such an evolution: Framelets form small architectural building blocks that can be easily understood, modified and combined. Reflection together with a high-level definition of semantic aspects allow the construction of partially self-configuring components. A case study corroborates that this constitutes a viable approach for rearchitecting legacy systems in practice.

1. PRODUCT LINE ARCHITECTURES FOR REPLICATED COMPONENTS

The source code of legacy systems comprises numerous replications of similar chunks of code. This means that from an architectural perspective many components of the overall architecture provide similar if not identical functionality. In other words, source code was written again and again from scratch for implementing these components. The idea for Rearchitecting legacy systems suffering from this problem is to develop a product line architecture for each such replicated component. The particular components are slight variations of the product line, that is, they belong to the family of the product line. Depending on the size of the replicated components, this kind of Rearchitecting activity will lead to a set of small product lines.

Let us take a look at a specific legacy system which we rearchitectured recently[1]. The three-tier client/server (CS) system of the bank is representative of legacy software systems relying on the CS architectural style. The clients (Windows PCs) access a central data repository via a remote procedure library, which is available as set of C functions. The data repository resides on a server machine (currently a Unix workstation; migration to Windows NT is under way) and/or a mainframe. The remote procedure library represents a quite stable part of the system architecture which has not changed at all over the past ten years. For implementing the client side the bank used a fourth-generation tool (SQL Windows/Gupta). The problem associated with this approach is that the tool produces a monolithic architecture: all dialog windows form one executable which has to be loaded to each client no matter how small the percentage of required dialogs actually is. From a development point of view it is hardly possible to package dialogs or parts of dialogs into reusable components.

From an architectural point of view the module structure of the client system is quite natural to envision. One dialog forms one module. In some cases a small group of dialogs might be packaged into one unit. Despite the shortcomings of typical fourth-generation tools regarding modularization, several other choices, such as state-of-the-art Java development environments allow the straightforward implementation of such a module structure.

A closer look at the module structure of dialogs reveals that almost every such module contains one or more components for handling remote procedure calls and one or more components for managing items in a list. Of course, the components differ in various contexts. For example, before a remote procedure is invoked, the input parameters of the procedure have to be read out of specific GUI elements. The number of parameters and the GUI elements differ between remote procedure calls (RPCs). RPCs return their results in C-arrays that have to be interpreted properly. The results are then displayed again in GUI elements. This infrastructure surrounding an

[1] The project is part of a cooperation between RACON Software GmbH, a software house of the Austrian Raiffeisen bank, and the Software Engineering Group at the University of Constance. The principal question at the outset was, whether a Rearchitecting effort based on framework technology and Java can lead to a significantly better modularization of the overall system that allows the reuse of components. Both aspects were defined as goals that could be achieved. The paper presents the concepts and ideas which we regard as generally useful for Rearchitecting legacy systems.

RPC is an example of source code that has to be implemented again and again for each RPC, but which offers similar functionality.

As most dialogs in real world CS systems have one or more GUI elements that display items in lists (by means of a GUI component called multi-column grid control), interactions associated with lists are also replicated in most dialogs. For example, a button for removing an item from the list has to be enabled only if an item in the list is selected, otherwise the button is disabled. Pressing a button to add an item opens a dialog window for entering the data. Pressing a button to modify an item also opens a dialog window and transfers the data representing the item to the corresponding GUI elements for the purpose of editing them. The aspects that differ in the various list-handling components are the types of the listed items, the dialog window to display an item, and some details such as button labels and the location of buttons for manipulating the list (for example, under the grid control or beside it).

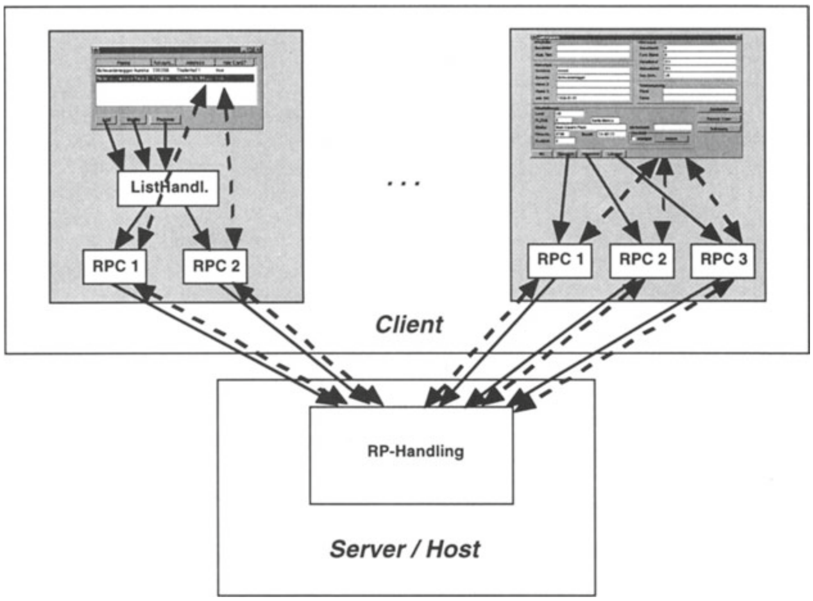


Figure 1: Module structure of the CS system with replicated RPC components

Figure 1 illustrates schematically the problem of replicated components in the architecture of the CS system at hand. Though the size of these components is small (about 200 to 300 source lines of code), they are replicated several hundred or even thousand times. Note that Figure 1 shows several replicated RPC components, but only one ListHandling component, as just one of the two sample dialogs contains a list. (Figures 1 and 2 apply the notation introduced by Bass et al. (1998). Solid arrows express control flow, dotted arrows depict data flow.)

Figure 2 shows an architectural solution which is based on a small product line for each such component. This solution is better as the number of components is significantly reduced.

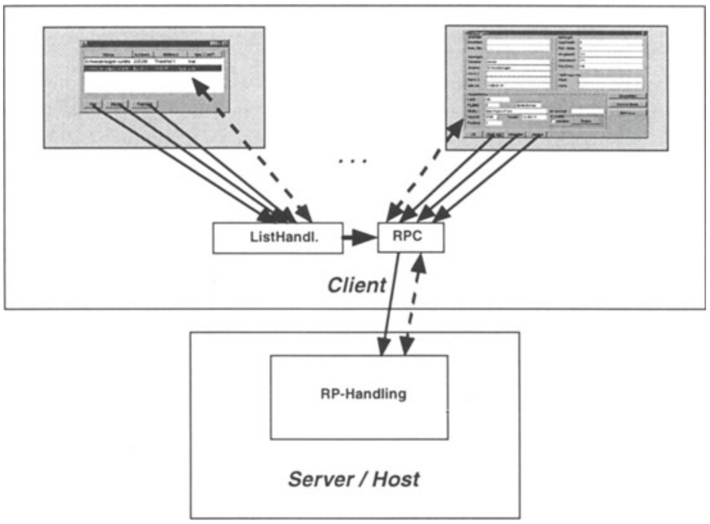


Figure 2: Module structure of the CS system with a product line architecture

The following sections present the concepts underlying reflection-based framelets. Such framelets were used to develop the product line architecture sketched above. A case study discussing the RPC framelet concludes the paper.

2. FRAMELETS

Object-oriented frameworks can be of any size, ranging from just one or a few simple classes to large sets of complex classes. However, the conventional idea of a framework is that it constitutes the skeleton of a complex, full-fledged application. Consequently, frameworks tend to be relatively large, consisting of, say, hundreds or thousands of classes. We argue that the common problems (see e.g., Casais (1995); Sparks et al. (1996); Bosch et al. (1998)) associated with frameworks stem from this idea.

We argue that the reason for common problems associated with frameworks is the conventional idea of a framework as the skeleton of a complex, full-fledged application:

- The design of such typical frameworks is hard. Due to the complexity and size of application frameworks and the lack of understanding of the framework design process, frameworks are usually designed iteratively, requiring substantial restructuring of numerous classes and long development cycles.
- Reuse of a framework is hard. A framework conventionally consists of the core classes of an application, and one has to understand the basic architecture of a particular application type to be able to specialize the framework.
- The combination of frameworks is hard. Often a framework assumes that it has the main control of an application. Two or more frameworks making this assumption are difficult to combine without breaking their integrity.

A framework becomes a large and tightly interconnected collection of classes that breaks sound modularization principles and is difficult to combine with other similar frameworks. Inheritance interfaces and various hidden logical dependencies cannot be managed by application programmers. A solution proposed by many authors is to move to black-box frameworks which are specialized by composition rather than by inheritance. Although this makes the framework easier to use, it restricts its adaptability. Furthermore, problems related to the design and combination of frameworks remain.

This suggests that it is not the construction principles of frameworks that form a problem, but the granularity of systems where they are applied. We propose a radical downsizing of frameworks and call these assets *framelets*. In contrast to a conventional framework, a framelet

- is small in size (< 10 classes),
- does not assume main control of an application, and
- has a clearly defined simple interface.

Like conventional frameworks, a framelet can be specialized by subclassing and composition.

We consider a framelet not only as a reusable asset but indeed as a fundamental unit of software in general. If a software system is seen as a set of service interfaces and their implementations, a framelet is any (small) subset of such a system (see Figure 3). An interface that belongs to the framelet without its implementation (and used within the framelet) is part of the specialization interface of the framelet. An interface that belongs to the framelet together with its implementation (and used outside of the framelet) is part of the service interface of the framelet. This is basically the foundation for using framelets in restructuring legacy systems.

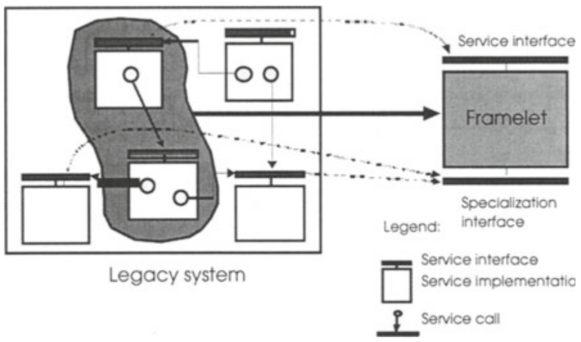


Figure 3: A framelet as a subset of a software system

Our vision is to have a family of related framelets for a domain area representing an alternative to a complex framework. Thus we view framelets as a kind of modularization means of frameworks. On a large scale, an application is constructed using framelets as black-box components, on a small scale each framelet is a tiny white-box framework.

A particular problem arising from the use of framelets as production lines is *specialization dependency*: the problem of specializing a large conventional framework may reappear in the case of framelets as the existence of various hidden dependencies that the specializations of individual framelets must follow to build a consistent application. Ideally, it should be possible to specialize each framelet independently of the others. To make this possible, framelets should be able to adapt themselves automatically to the context in which they are being used, relieving the programmer of the burden of explicitly writing the context requirements as configuration code in the specialization. In the sequel we show that this can be at least partially achieved using reflective features provided by many OO languages (e.g., Java), together with certain semantic conventions.

3. REFLECTION AS THE BASIS OF SELF-CONFIGURING ASSETS

What frameworks and framelets have in common is that they represent one means of implementing product line architectures. For this purpose they rely on the constructs provided by object-oriented programming languages. The few essential framework construction principles, as described, for example, by Pree (1996), are applicable to framelets as well. A framelet retains the call-back characteristic (a.k.a. the Hollywood principle) of white-box frameworks: framelets are assumed to be extended with application-specific code called by the framelet. Figure 4 (a) shows a run-time snapshot of a framelet with the objects A and B as hot spots. Usually hot spots correspond to abstract classes or Java interfaces in the static program code. A reuser of the framelet would have to choose either from already existing specific subclasses of the abstract classes or from interface implementations, or would have to implement appropriate classes. The framelet is adapted by replacing the place holders by instances of specific A and B classes (see Figure 4 (b)).

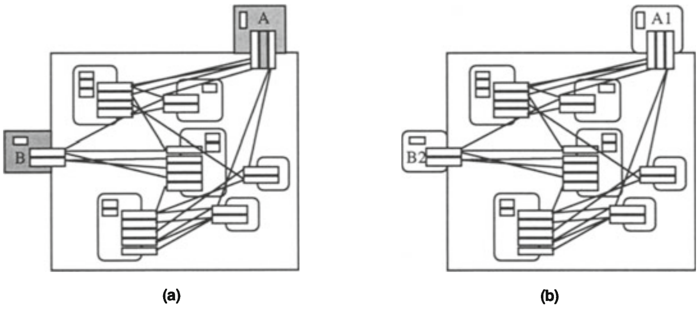


Figure 4: Framelet before (a) and after (b) adaptation

Besides the mentioned canonical possibilities of defining abstract entities of a framelet, there exist significantly more flexible ways of doing this, albeit ones that sacrifice type safety. Let us assume we design the framelet sketched in Figure 4 in Java, where all classes have a common ancestor, i.e., they inherit from class Object. Now the framelet designer could decide not to restrict the two hot spots to a specific type, such as A and B in our example. Instead it should be possible to plug any object into the framelet. In other words, the static type of these hot spots becomes Object. The only useful assumption that the framelet designer can make about these abstract entities is that they provide the full range of meta-information. As meta-information is supported by the Java standard library (JDK 1.1 and above) any object offers the same range of meta-information. For example, it becomes possible to iterate over instance variables, access their types and values, iterate over an object's methods and invoke particular ones. On first consideration, this seems to be useless: No semantics are associated with these operations, as opposed to abstract classes or interfaces, whose methods define a specific type with an associated behavior on which the framework developer can rely.

The advantage of such reflection-based hot spots is that somewhat "intelligent" framelets can be constructed that exhibit self-configuring properties. The framelet generically couples itself with the objects that fill the hot spots. In order to make this happen, some semantics have to be defined for the abstract entities. The sample framelet discussed in the next section applies a very simple mechanism for defining semantics, i.e., naming conventions. The point is that the semantic definitions are completely decoupled from the programming language level. They reintroduce a notion of typing on a more domain-related level. Thus proper semantic definitions render void the above mentioned drawback of giving up strong typing. They introduce kinds of equivalents of types on the domain level. Of course, naming conventions are probably the most basic means of defining semantics. We are currently investigating more sophisticated means of pragmatically defining domain-specific semantics.

4. THE RPC PRODUCT LINE—A CASE STUDY

Remember that calling a remote procedure requires some infrastructure in addition to the mere invocation. The values of the input parameters of the remote procedure originate from GUI elements. The return parameters of most remote procedures are packaged in a C-array that has to be carefully processed before they can be displayed in particular GUI elements.

The interface of a reusable asset should be designed as straightforwardly for the user as possible. If the infrastructure surrounding an RPC is packaged in a reusable asset, the ideal situation would be that the reuser just invokes one method, doRPC(...), of this component. The first parameter is the name of the RPC as a string. The second parameter of doRPC(...) is a reference to the dialog window which contains the GUI elements corresponding to the input parameters of the remote procedure. Finally, a reference to the dialog window has to be specified in whose GUI elements the result parameter values are displayed. Let's call these two dialogs input and output dialog windows. Note that the input and output dialog windows can be identical. The RPC component should ideally be able to do the configuration job itself, i.e., extract the parameter values from the appropriate GUI elements of the input dialog window and transfer the results to the GUI elements of the output dialog window. This would make the reusable asset a perfect small product line for calling remote procedures. How can such a convenient reuse level be achieved?

Here a simple naming convention comes into play. The GUI elements have to have the same names as the RPC parameters. The RPC component is implemented as a framelet in Java with two core hot spots: the input dialog window and the output dialog window. Both hot spots are of type Object. As discussed in detail below, the RPC framelet only requires the meta-information interface to accomplish the configuration job. We'll see that the naming convention is a sufficient semantic specification of the behavior of the two hot spots.

The RPC product line works internally as follows: The framelet is based on a parameter description for each remote procedure. The type of each parameter of a particular remote procedure has to be known. Furthermore, a parameter has to be classified as an input or an output parameter. In the realm of the RPC framelet, the class construct was chosen to describe a remote procedure. (These classes don't have to be written by hand. A tool generates these descriptions out of the available RPC documentation.) Each such class contains besides an empty constructor only public instance variables. The instance variables correspond to the parameters of the remote procedure. The instance variable names reflect the parameter names in the remote procedure documentation. A suffix Out marks output parameters. The types of the instance variables correspond to the types of the remote procedure parameters.

In order to call a remote procedure, including all the data fetching and processing that is associated with a call, the reuser sends the message doRPC(...) to the RPC framelet, passing the remote procedure name as well as the input and output dialog variables as parameters as sketched above.

Based on the remote procedure name, the RPC framelet first searches the class that describes the parameters of the remote procedure, and instantiates this class. The framelet then iterates over the instance variables of this object and assigns to them those values to them which it retrieves from the GUI elements of the input dialog window that have the same name as the parameters in the description object. For this purpose, the framelet iterates over the instance variables of the dialog window. This works fine as the GUI elements of a dialog window manifest in public instance variables of that dialog window object.

Figure 5 exemplifies the interaction between various components during the invocation of a remote procedure. The solid lines again depict control flow, whereas the dotted lines represent data flow. Activating the button Search ("Suchen" in the dialog window with German labels) should imply the invocation of a remote procedure SearchPerson which basically searches all records in a database that correspond to the search parameters (e.g., the entered last name).

First the method doRPC(...) is called (label 1 in Figure 5) and receives the following parameters: the name of the remote procedure as a string, and the references to the input and output dialog windows. In this example both refer to the same dialog. The RPC framelet now retrieves the values from the input dialog window in order to assign these values to the remote procedure parameter description object (a) and calls the remote procedure (labels 2 and 3 in Figure 5).

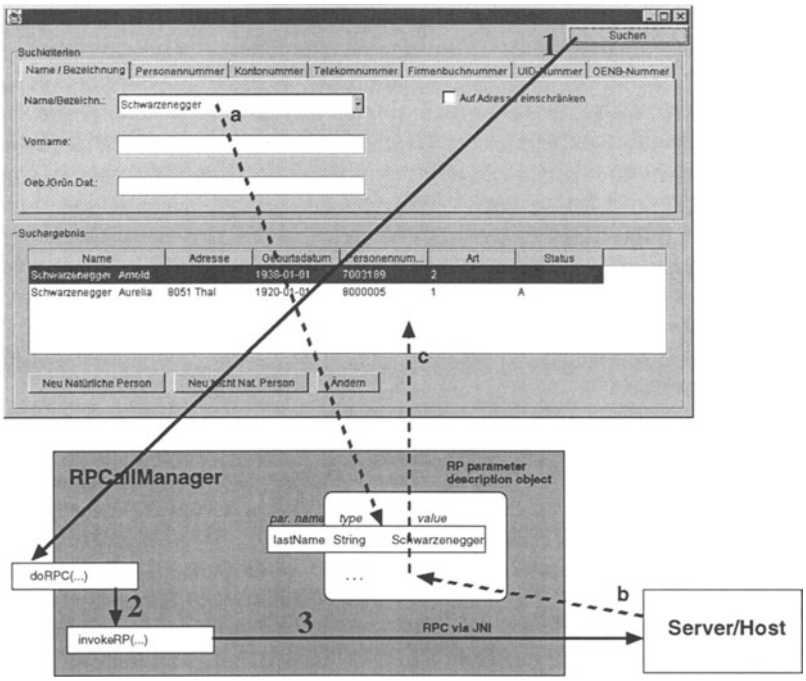

Suchen
1
Suchkriterien
Name / Bezeichnung | Personennummer | Kontonummer | Telekomnummer | Firmenbuchnummer | UID-Nummer | OENB-Nummer
Name/Bezeichn.: Schwarzenegger
a
Auf Adresse einschränken
Vorname:
Geb./Grün.Dat.:
Suchergebnis
Name | Adresse | Geburtsdatum | Personennum... | Art | Status
Schwarzenegger Arnold 1938-01-01 7003189 2
Schwarzenegger Aurelia 8051 Thal 1920-01-01 8000005 1 A
Neu Natürliche Person
Neu nicht Nat. Person
Ändern
c
RPCallManager
RP parameter description object
par. name type value
lastName String Schwarzenegger
...
doRPC(...)
2
invokeRP(...)
3
RPC via JNI
b
Server/Host

Figure 5: Schematic representation of interactions and data flow in a RPC framelet adaptation

Note that the name of the GUI element which displays the string Schwarzenegger is not visible in Figure 5. The GUI element has the internal[2] name *lastName* and thus adheres to the naming convention. One instance variable of the remote procedure description object also has the name *lastName*.

The RPC framelet finally processes the results returned from the host or server and assigns the values to the proper instance variables of the remote procedure description object (b). The remote procedure description object provides some additional information how to process the result (a C-array structure) for each remote procedure. This detail will not be discussed in this paper. From there the RPC framelet transfers the data via the meta-information interface and naming convention into the GUI elements of the output dialog window (c).

The source code in Example 1 illustrates how reflection allows the generic implementation of a RPC. The second parameter of this method is the remote procedure description object whose role is explained above.

The classes Class, Method and Field are part of the standard Java library. The first line of method invokeRP(...) stores all instance variables (fields in Java jargon) of the remote procedure description object in an array. Suppose that the object has N instance variables, then the arrays, params, and args have the initial size N=fields.length. The for-loop assigns the particular array component the type (params[i]= fields[i].getType()) and the value of the instance variable (args[i]= fields[i].get(parametersOfRPC)).

The class ListOfRPCs contains all remote procedures as methods. The methods invoke the associated C functions by means of the Java Native Interface (JNI). The statement getMethod(...) returns the Method object that corresponds to the name of the remote procedure. This is the first parameter of method invokeRP(...). The reference to this Method object is stored in the variable RPCmethod. Class Method offers a method invoke(...) to finally carry out the call.

The selected source code illustrates how reflection is useful to decouple the framelet from the specific remote procedure library. The description of a remote procedure in a separate class suffices for a generic implementation of a remote procedure call in the framelet. New or changed remote procedures only require additional or modified descriptions. The RPC framelet itself is not affected.

```
class RPCallManager ... {
```

[2] The GUI editor assigns a name to each GUI element. A tool generates Java code which corresponds to the visual/interactive specification of the GUI. In general, a dialog window is represented in one class. The GUI elements contained in a dialog window become instance variables of this class. The GUI element names determine the instance variable names.

```
        ListOfRPCs rpcList;  // contains all RPCs as methods
        ...
        RPCallManager (...)  {
              ...
        }
        ...
        public void doRPC(String RPCname, Object inDialog, Object outDialog) {
              ...
        }
        protected void invokeRP(String nameOfRPC,
                                          Object parametersOfRPC)  {
            Field[] fields = parametersOfRPC.getClass().getDeclaredFields();
            Method RPCmethod = null;       // auxiliary var. for invoking RPC
            Class[] params = new Class[fields.length];
            Object[] args = new Object[fields.length];

            for all params do {
                  params[i] = fields[i].getType();  // type of parameter
                  try {
                        args[i] = fields[i].get(parametersOfRPC);  // par. value
                  } catch (IllegalAccessException iae) { ... }
            }
            RPCmethod = rpcList.getClass().getMethod(nameOfRPC, params);

            ... // exception handling
            RPCmethod.invoke(args);
            ... // exception handling
       }
  }
```

Example 1: Generic implementation of the remote procedure call

Overall, the automated configuration of the RPC product line relies solely on meta-information. A method of class Class called newInstance() allows the instantiation of a class whose name is provided as string. Class Class also offers methods for iterating over the instance variables of an object. Both properties together are sufficient for the implementation of the RPC framelet.

Measurements of the run-time overhead of iterating over instance variables showed that the overhead can be neglected. The time for generically assembling an RPC takes between 0.2 and 0.5% of an RPC.

5. CONCLUSION

We have introduced two basic concepts for extracting reusable elements from legacy systems: framelets and dynamic specialization through reflection. The latter mechanism supports the idea of a framelet by automating part of the specialization work. Neither of these concepts is strictly limited to the OO world, but our discussion and case study have been carried out in the context of OO: this paradigm fits well our purposes through its mechanisms for abstraction, specialization and reflection. To some extent, corresponding mechanisms are provided by various component technologies (e.g., COM).

It should be emphasized that dynamically configurable framelets are not only useful for restructuring legacy systems, but they can and should be used as basic architectural units in the design of new systems as well. Since a framelet implements only a restricted functionality, its development is expected to be far less iterative than the development of a typical application framework. Hence, a mature generic software system based on framelets can be developed in essentially shorter time than a conventional framework, yet retaining the applicability of a framework.

The feasibility of framelets may depend on the overall architectural style. It seems that framelets are particularly natural units in a layered architecture where the services required by a layer are implemented by a lower layer. In this case a single layer can be sliced into several framelets. For each such slice, the interface to the upper layer represents the specialization interface while the interface to the lower layer represents the service interface of the framelet.

Though framework-related design patterns (Gamma et al., 1995; Buschmann et al., 1996) represent architectural knowledge, they are too small to become the foundation of reusable architectural components. Based on the first experience with framelets we argue that framelets might be a pragmatic compromise between design patterns and application frameworks. Framelets might be viewed as the combination of a few design patterns into a reusable architectural building block.

To which degree an application can be based on framelets remains an open question, but we feel that frequently used independent features suitable for framelets can be easily found in many application domains. Future work will focus on the prototypical development of framelet families, on investigation of pragmatic semantic conventions used for the automatic configuration of framelets, and on programming tools supporting the use of framelets.

REFERENCES

Bass L., Clements P., Kazman R. (1998) Software Architecture in Practice. Addison-Wesley 1998.

Bosch, J, Mattsson, M., and Fayad, M. (1998): Framework Problems, Causes, and Solutions, CACM, 1998 (will appear)

Buschmann F., Meunier R., Rohnert H., Sommerlad P. and Stal M. (1996) *Pattern-Oriented Software Architecture—A System of Patterns*. Wiley and Sons

Casais E. (1995): An Experiment in Framework Development. *Theory and Practice of Object Systems* 1, 4(1995), 269-280.

Fayad, M. and Schmidt, D (1997) Object-Oriented Application Frameworks. CACM, Vol. 40, No. 10, October 1997.

Gamma E., Helm R., Johnson R. and Vlissides J. (1995). *Design Patterns—Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley

Pree W. (1996). *Framework Patterns*. New York City: SIGS Books (German translation, 1997: *Komponentenbasierte Softwareentwicklung mit Frameworks*. Heidelberg: dpunkt)

Pree W. and Koskimies K. (1998): Framelets—Small and Loosely Coupled Frameworks. ACM Symposium on Frameworks (will appear)

Sparks S., Benner K., Faris C. (1996): Managing Object-Oriented Framework Reuse. Computer 29,9 (Sept 96), 52-62.

Checking the Correctness of Architectural Transformation Steps via Proof-Carrying Architectures

R. A. Riemenschneider
Computer Science Laboratory, SRI International, Menlo Park, CA, USA
rar@csl.sri.com

Key words: Software architectures, architecture hierarchies, transformation, refinement verification, proof-carrying architectures

Abstract: The end product of architecting is an *architectural hierarchy*, a collection of architectural descriptions linked by mappings that interpret the more abstract descriptions in the more concrete descriptions. Formalized transformational approaches to architecture refinement and abstraction have been proposed. One argument in favor of formalization is that it can result in architectural implementations that are guaranteed to be correct, relative to the abstract descriptions. If these are correct with respect to one another, conclusions obtained by reasoning from an abstract architectural description will also apply to the implemented architecture. But this correctness guarantee is achieved by requiring that the implementer use *only* verified transformations, i.e., ones that have been proven to produce correct results when applied. This paper explores an approach that allows the implementer to use transformations that have not been proven to be generally correct, without voiding the correctness guarantee. *Checking* means determining that application of the transformation produces the desired result. It allows the use of transformations that have not been generally verified, even ones that are known to sometimes produce incorrect results, by showing that they work *in the particular case*.

1. INTRODUCTION

The process of specifying an architecture often begins by providing a very high-level description of it. This description characterizes the architecture in terms of a few abstract components, perhaps the principal functions the system must perform and some data stores. These components are linked by abstract connectors, perhaps indicating dataflow or control flow relationships among the components. This abstract description provides an easily understood overview of the entire system architecture, but omits so much detail that it provides relatively little guidance to someone charged with implementing the architecture using programming-language-level and operating-system-level constructs. So the abstract description must be successively refined—with complex components and connectors decomposed into simpler parts, and abstract specifications of operations and relationships replaced by more concrete specifications—until an appropriate amount of detail has been added. It usually is desirable to continue the refinement until implementation-level constructs have replaced all the abstractions.

Alternatively, architecting a system can consist of assembling instances of reusable component and connector types selected from a library. Such libraries effectively make the implementation-level architecture more abstract, and reduce the conceptual gap between the requirements specification and the implemented architecture. Nevertheless, combining a large number of components and connectors in complex ways can easily result in an architecture that is hard to understand and analyze. So, it is desirable to generate more easily comprehensible abstract representations of the implementation-level architecture.

In either case, the end product of the architecting process is typically a collection of architectural descriptions, at different levels of abstraction and often in different styles (Garlan, Allen, & Ockerbloom 1994). The more abstract descriptions are linked to the more concrete descriptions by interpretation mappings. An interpretation mapping says how the abstractions are implemented.[1] It sends each sentence in the language of the abstract description to a corresponding sentence in the language of the concrete description. For example, the fact that some component a is implemented by components a_1, a_2,…., a_n would be indicated by mapping the sentence

$$\mathsf{Component(a)}$$

to the sentence

$$\mathsf{Component(a_1)} \wedge \mathsf{Component\ (a_2)} \wedge \ldots \wedge \mathsf{Component(a_{\mathit{n}})}$$

[1]For more details on characterizing implementation steps using interpretation mappings, see our earlier paper (Moriconi, Qian, & Riemenschneider 1995).

The collection of architectural descriptions and interpretation mappings that comprise the complete architectural specification is called an *architecture hierarchy*.

There are many advantages to formalizing refinement and abstraction in system development: a library of refinement or abstraction transformations provides a "corporate knowledge base" of standard, or preferred, development patterns; mechanizing the application of these transformations lessens the likelihood of clerical errors during the development process; reuse of the transformations will result in greater validation of the patterns they codify; and so on. But one of the most fundamental advantages of formalization is that it allows the average developer to produce abstraction hierarchies that are guaranteed to be consistent. In other words, the use of verified transformations in the development process will guarantee that abstractions accurately characterize implementations, albeit more abstractly. A verified refinement transformation is one that has been proven to produce a correct implementation of whatever it is applied to. A verified abstraction transformation is one that has been proven to produce a correct abstraction of whatever it is applied to.

Even if attention is restricted to the case of architectures, there is some debate as to exactly what *correct* should mean. We have proposed a somewhat stricter-than-usual criterion for correctness (Moriconi, Qian & Riemenschneider 1995), while others have argued that the standard criterion is preferable (Philipps & Rumpe 1997). For present purposes, any reasonable criterion that characterizes correctness in terms of preservation of truth will do perfectly well. The standard correctness criterion is that every consequence of the abstract description must be a consequence of the concrete description as well. More precisely, for every sentence **A** in the language of the abstract description, where T_1 is the logical theory that formalizes the abstract description,

$$T_1 \vdash \mathbf{A} \Rightarrow T_2 \vdash \mu(\mathbf{A})$$

where T_2 is the theory that formalizes the concrete description, and μ is the interpretation mapping that links the two theories.[2] A mapping μ that satisfies this condition is called an *interpretation of T_1 in T_2*. Our proposed stronger criterion for purely structural descriptions replaces the conditional with a biconditional, i.e., requires that the interpretation mapping be a *faithful theory interpretation*. One might also employ weaker-than-standard

[2]Our earlier paper explains how to formalize structural descriptions of architectures as logical theories. Since structural descriptions are largely declarative, the process is quite straightforward.

criteria, where only some consequences of the theory—properties of special interest—need be preserved.

What all these criteria have in common is that they justify the use of formal reasoning about the architecture based on the more abstract descriptions. If some sentence is shown to be a formal consequence of the abstract architectural theory, the concrete theory is known to correctly implement the abstract theory, and the sentence is among those that the correctness criterion guarantees are preserved by the implementation, then the sentence is known to be a consequence of the concrete theory as well. It is correctness guarantees that link the results of abstract analyses to the real world.

The usual approach to producing a correctness guarantee is restricting the architect to the use of verified transformations. This approach suffers from a problem, in practice. Even given a fairly mature library of verified transformations, it would hardly be surprising if an architect found himself unable to perform a certain refinement or abstraction step that he believed to be correct because the required transformation has not been included in the library. Expecting the typical system architect to produce a formal proof that the step is correct is unrealistic, yet the presence of a single unverified implementation step in the hierarchy voids the correctness guarantee provided by the restriction to verified transformations. Is there any way to allow the user to include such arbitrary steps in the development of the architecture hierarchy, while maintaining a correctness guarantee?

2. PROOF-CARRYING ARCHITECTURES

Our solution to this problem is based on the notion of checking the correctness of steps in architecture hierarchy development. By *checking*, we mean automatically performing some calculation that shows the step is correct. Checking can be substantially simpler than verification, because it is focussed on a particular step. Verifying a transformation means showing that it *always* produces correct results, while checking a transformation step means showing that a correct result was obtained *in one specific case*. Thus, checking entirely avoids the sometimes difficult problem of characterizing the preconditions required for the transformation to produce correct results (Riemenschneider 1998).

Our initial approach to checking transformation steps was inspired by work on compilers that generate proof-carrying code (PCC) (Necula & Lee 1998). The basic idea is that, rather than attempting to prove the transformations performed by a compiler always produce code with certain desired properties, to generate a purported formal proof that the complied code has those properties as part of the code generation process. The purported proof can then be checked and, if it turns out to be a correct proof, it follows that the generated code has the desired properties. Thus, the emphasis is shifted from showing that compiler transformations are correct in general to checking that they produced correct results in individual cases.

The application of this idea to architectural transformation is straightforward. At some abstract level, the architectural description is proven to guarantee that the architecture has some desirable property, **C**. The interpretation mapping μ that sends abstract level sentences to their implementations can also be applied to the proof of **C**. If the image of the proof under the implementation mapping turns out to be a correct proof that the implementation has $\mu(\mathbf{C})$, then, of course, the implementation has $\mu(\mathbf{C})$. Checking the transformed proof can, therefore, provide the desired correctness guarantee.

3. AN EXAMPLE: SECURE DISTRIBUTED TRANSACTION PROCESSING

The idea of proof-carrying architectures can be illustrated by an example, based on our development of software architectures for secure distributed transaction processing (SDTP) (Moriconi, Qian, Riemenschneider & Gong 1997). These architectures extend X/Open's standard DTP architecture (X/Open Company 1993) by enforcing a simple "no read up, no write down" security policy. The primary result of our development efforts is a hierarchy that links an extremely abstract architectural description, shown in Figure 1, to three implementation-level descriptions written in a style that can be directly translated into a programming language such as Java using standard network programming constructs. The gap between the abstract SDTP architecture and each concrete SDTP architectures is filled by roughly two dozen descriptions—the exact number varies among the implementations—at intermediate levels of abstraction, linked in a chain by interpretation mappings.

We are in the process of formally proving the implementation-level architectures are secure by proving that the abstract description is secure, and proving that every interpretation mapping preserves security. One of the techniques that is being employed is showing that the interpretations incrementally transform the abstract-level security proof into implementation-level security proofs. The example below shows how the interpretation mapping associated with the first refinement step in all three chains transforms the abstract security proof into a slightly more concrete security proof.

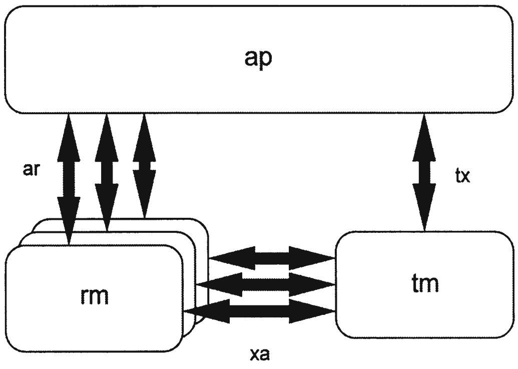


Figure 1. Abstract SDTP architecture — components linked by secure channels

3.1 The abstract SDTP architecture

Figure 1 depicts the most abstract architecture for SDTP. The boxes are the components of the architecture: the Application (labeled "ap"), some number of Resource Managers (labeled "rm"), and a Transaction Manager (labeled "tm"). The components are linked by *secure channels*, indicated by the heavy double headed arrows that make up the interfaces between the Application and Resource Managers, the Application and the Transaction Manager, and the Resource Managers and the Transaction Manager. Secure channels are a type of connector that enforce the security policy. In other words, secure channels will not carry classified data from a component to a component that lacks required clearances. To say that the system as a whole satisfies the security policy means that there is no flow of classified data to a component that lacks the required clearances.

3.2 An abstract proof of security

Informally, the security of the system follows almost immediately from the fact that it employs only secure channels. Not surprisingly, a textbook-style natural deduction proof (Lemmon 1987, Mates 1972) of system security is quite simple[3]. Consider the dataflow from some given Resource Manager rm to the Application ap, for example. A proof of the formula

[3] In this paper, I will use natural deduction, since that provides a familiar concrete representation of formal proofs. In our actual verifications that the SDTP hierarchy's interpretation steps preserve security, we are employing the PVS verification system [18].

$$(\forall d : \mathsf{Labeled\_Data})\ [\mathsf{Flows}(d, rm, ap) \supset \mathsf{label}(d) \geq \mathsf{clearance}(ap)]$$

which says

> every labelled datum d that flows from rm to ap has a security label classifying it that is less than or equal to the clearance level of ap

from five axioms of the architectural theory is shown in Figure 2.

{1}	1.	$(\forall d : \mathsf{Labeled\_Data})[\mathsf{Flows}(d, rm, ap) \supset \mathsf{Carries}(secure\_ar\_channel, d, rm's\_ar\_port, ap's\_ar\_ports(rm))]$	Axiom describing specific architecture
{2}	2.	$\mathsf{Port\_Of}(ap's\_ar\_ports(rm), ap)$	Axiom describing specific architecture
{3}	3.	$(\forall c : \mathsf{Secure\_Channel})(\forall d : \mathsf{Labeled\_Data})(\forall x : \mathsf{Output\_Port})(\forall y : \mathsf{Input\_Port})[\mathsf{Carries}(c, d, x, y) \supset \mathsf{label}(d) \leq \mathsf{clearance}(y)]$	Axiom characterizing secure channels
{4}	4.	$(\forall a : \mathsf{Component})(\forall y : \mathsf{Input\_Port})[\mathsf{Port\_Of}(y, a) \supset \mathsf{clearance}(y) \leq \mathsf{clearance}(a)]$	Axiom constraining port clearances
{5}	5.	$(\forall s_1, s_2, s_3 : \mathsf{Security\_Label})[s_1 \leq s_2 \wedge s_2 \leq s_3 \supset s_1 \leq s_3]$	Axiom specifying transitivity of security label ordering
{1}	6.	$\mathsf{Flows}(d_0, rm, ap) \supset \mathsf{Carries}(secure\_ar\_channel, d_0, rm's\_ar\_port, ap's\_ar\_ports(rm))$	Universal instantiation (1)
{3}	7.	$\mathsf{Carries}(secure\_ar\_channel, d_0, rm's\_ar\_port, ap's\_ar\_ports(rm)) \supset \mathsf{label}(d_0) \leq \mathsf{clearance}(ap's\_ar\_ports(rm))$	Universal instantiation (3)
{1, 3}	8.	$\mathsf{Flows}(d_0, rm, ap) \supset \mathsf{label}(d_0) \leq \mathsf{clearance}(ap's\_ar\_ports(rm))$	Tautological consequence (6,7)
{4}	9.	$\mathsf{Port\_Of}(ap's\_ar\_ports(rm), ap) \supset \mathsf{clearance}(ap's\_ar\_ports(rm)) \leq \mathsf{clearance}(ap)$	Universal instantiation (4)
{2, 4}	10.	$\mathsf{clearance}(ap's\_ar\_ports(rm)) \leq \mathsf{clearance}(ap)$	Tautological consequence (2,4)
{5}	11.	$\mathsf{label}(d_0) \leq \mathsf{clearance}(ap's\_ar\_ports(rm)) \wedge \mathsf{clearance}(ap's\_ar\_ports(rm)) \leq \mathsf{clearance}(ap) \supset \mathsf{label}(d_0) \leq \mathsf{clearance}(ap)$	Universal instantiation (5)
{2, 4, 5}	12.	$\mathsf{label}(d_0) \leq \mathsf{clearance}(ap's\_ar\_ports(rm)) \supset \mathsf{label}(d_0) \leq \mathsf{clearance}(ap)$	Tautological consequence (10,11)
{1, 2, 3, 4, 5}	13.	$\mathsf{Flows}(d_0, rm, ap) \supset \mathsf{label}(d_0) \leq \mathsf{clearance}(ap)$	Tautological consequence (8,12)
{1, 2, 3, 4, 5}	14.	$(\forall d : \mathsf{Labeled\_Data})[\mathsf{Flows}(d, rm, ap) \supset \mathsf{label}(d) \leq \mathsf{clearance}(ap)]$	Universal generalization (13)

Figure 2. Formal proof that dataflow from rm to ap satisfies the security policy

The five axioms say
1. every labelled datum d that flows from rm to ap is carried by secure_ar_channel form the output port rm's_ar_port to the input port of the port array ap's_ar_ports that is indexed by rm,
2. the input port of the port array ap's_ar_ports that is indexed by rm is a port of ap,
3. if secure channel c carries labelled datum d from output port x to input port y, then d's security label is less than or equal to the clearance level of y,

4. the clearance level of any port y of component a must be less than or equal to the clearance level of a, and
5. the ordering of security labels is transitive.

The first two axioms are facts about the particular architecture, the third axiom is the defining property of the secure channel subtype, the fourth and fifth axioms are general axioms of the security model.

3.3 A slightly more concrete SDTP architecture

The secure channels of abstract SDTP architecture can be implemented in terms of ordinary dataflow channels and additional components in a variety of ways, depending upon the security properties of the components (Moriconi, Qian, Riemenschneider & Gong 1997). The most interesting implementation is shown in Figure 3, where the light double headed arrows represent ordinary dataflow channels that do not enforce the security policy.

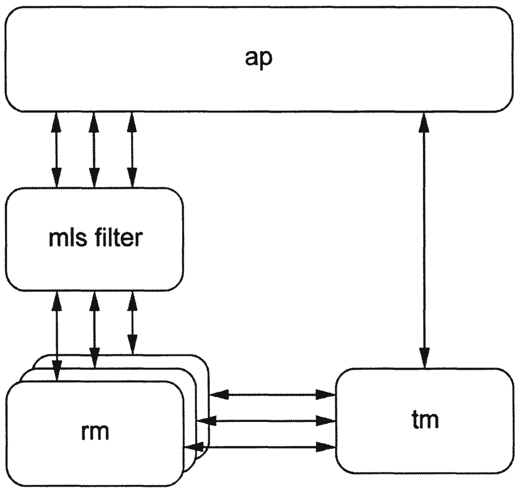


Figure 3. More concrete SDTP architecture—secure channels refined to ordinary channels, or ordinary channels plus security filter

This implementation is suited to the case where all of the resource managers are single-level, but not necessarily the same level. The security policy is enforced by a multi-level secure component that filters dataflow between the application and the resource managers: if passing a datum from a resource manager to the application would violate the security policy, the filter removes it from the stream.

The concrete architecture can be thought of as resulting from the abstract architecture by applying several transformations. For example, one transformation, the *Filter Introduction Transformation* (*FIT*), replaces secure channels between components that are not multilevel secure by ordinary dataflow channels and a component that enforces the security policy.

3.4 A slightly more concrete proof of security

Now it must be shown that, like the abstract SDTP architecture, the more concrete SDTP architecture has the desired security property. The two conventional approaches to establishing this result are

1. to directly prove that the more concrete architecture is secure, in much the same way the abstract architecture was proven secure (perhaps using the abstract-level proof for heuristic guidance), and
2. to show that the Filter Introduction Transformation (FIT), and the other transformations that produce the more concrete architecture from the abstract architecture, always preserves the security properly.

The use of proof-carrying architectures provides a third alternative.

When transformation FIT is applied, it can be applied not only to the architectural description, but to the formal security proof of Figure 2 as well. The result of applying FIT to this proof is shown in Figure 4, where the implementation mapping μ associated with this application is determined as follows. A complete account of how first-order interpretation mappings are defined, and basic facts about them, can be found in logic textbooks (Enderton 1972, Shoenfield 1967)[3]. For present purposes, it is enough to know that

1. for every term $\mathbf{t}$ of the language of the abstract theory, $\mu(\mathbf{t})$ is a (possibly complex) term of the language of the more concrete theory,
2. for every predicate $\mathbf{F}$ of the language of the abstract theory, $\mu(\mathbf{F})$ is a (possibly complex) predicate of the language of the more concrete theory,
3. for every formula $\mathbf{A}$ of the language of the abstract theory,
$$\mu(\neg\mathbf{A}) = \neg\mu(\mathbf{A})$$
and similarly for the other connectors, and
4. for every formula $\mathbf{A}$ of the language of the abstract theory, every variable $\mathbf{x}$, and every type predicate $\mathbf{T}$ of the language of the abstract theory,
$$\mu((\forall\mathbf{x} : \mathbf{T})\ \mathbf{A}) = \mu(\forall\mathbf{x} : \mathbf{T})\ \mu(\mathbf{A})$$
where $\mu(\forall\mathbf{x} : \mathbf{T})$ is a sequence of universal quantifiers, and similarly for the other quantifiers.

[3] Technically, we will make use of what are called n-dimensional interpretations (Hodges 1993, pp. 212.) But this is a reasonably straightforward generalization of the definition found in the cited textbooks.

The Carries predicate

$$\mathsf{Carries}(\langle \textit{secure channel}\rangle, \langle \textit{datum}\rangle, \langle \textit{out port}\rangle, \langle \textit{in port}\rangle)$$

that is mentioned in formulas 1, 3, 6, and 7 of the abstract-level proof is mapped to a conjunction of the Carries, Passes, and Carries predicates,

$$\mathsf{Carries}(\langle \textit{channel}\rangle, \langle \textit{datum}\rangle, \langle \textit{out port}\rangle, \langle \textit{filter in port}\rangle)$$

$$\wedge\ \mathsf{Passes}(\langle \textit{filter}\rangle, \langle \textit{datum}\rangle, \langle \textit{filter in port}\rangle, \langle \textit{filter out port}\rangle)$$

$$\wedge\ \mathsf{Passes}(\langle \textit{channel}\rangle, \langle \textit{datum}\rangle, \langle \textit{filter out port}\rangle, \langle \textit{in port}\rangle)$$

This clause in the definition of μ says that a secure channel carrying a datum from some output port to some input port is implemented as a channel carrying the datum from the output port to some input port of a filter, passing the datum through the filter from the input port to some output port, and carrying the datum from output port of the filter to the input port[5]. This mapping is also applied to formula 3 in order to preserve the fact that formula 7 should follow from formula 3 by Universal Instantiation.

The universal quantifier over secure channels in formula 3,

$$(\forall\ \langle \textit{secure channel variable}\rangle : \mathsf{Secure\_Channel}\rangle$$

is mapped by μ to universal quantifiers over channels and a universal quantifier over MLS components,

$$(\forall\ \langle \textsf{\textit{to-filter channel variable}}\rangle \text{: } \mathsf{Channel}\rangle$$

$$(\forall\ \langle \textsf{\textit{filter variable}}\rangle : \mathsf{MLS\_Component}\rangle$$

$$(\forall\ \langle \textsf{\textit{from-filter channel variable}}\rangle : \mathsf{Channel}\rangle$$

It is easy to check that the result of applying the FIT interpretation mapping μ to the proof of security is a syntactically correct derivation of the

[5] This mapping would not be appropriate to apply to every occurrence of the Carries predicate in every derivation, because some secure channels in the abstract architecture may not be replaced by a combination of two channels and a filter in the concrete architecture. However, formulas 1, 6, and 7 of the proof specifically refer to what secure_ar_channel carries, and this secure channel *is* being implemented by two channels and a filter, so I will use this simpler interpretation for purposes of the example.

desired security property from formulas that are images of axioms of the more abstract architectural theory. Mapping μ sends tautological consequence steps to correct tautological consequence steps, universal instantiation steps to correct universal instantiation steps, and universal generalization steps to correct universal generalization steps. So μ has indeed mapped the formal abstract-level security proof to a concrete-level security proof, but not necessarily a proof *from axioms of the concrete architectural theory*.

$$
\begin{array}{rrl}
\{1\} & 1. & (\forall d : \text{Labeled\_Data})[\text{Flows}(d, rm, ap) \\
 & & \qquad \supset \text{Carries}(\text{rm\_to\_filter\_channel}, d, \text{rm's\_ar\_port}, \text{filter\_in\_port}(rm))] \\
 & & \qquad\qquad \wedge \text{Passes}(\text{mls\_filter}, d, \text{filter\_in\_port}(rm), \text{filter\_out\_port}(rm))] \\
 & & \qquad\qquad \wedge \text{Carries}(\text{filter\_to\_ap\_channel}(rm), d, \text{filter\_out\_port}(rm), \text{ap's\_ar\_ports}(rm))] \\
\{2\} & 2. & \text{Port\_Of}(\text{ap's\_ar\_ports}(rm), ap) \\
\{3\} & 3. & (\forall c_1 : \text{Channel})(\forall f : \text{MLS\_Component})(\forall c_2 : \text{Channel})(\forall d : \text{Labeled\_Data}) \\
 & & \quad (\forall x_1 : \text{Output\_Port})(\forall x_2 : \text{Output\_Port})(\forall y_1 : \text{Input\_Port})(\forall y_2 : \text{Input\_Port}) \\
 & & \qquad [\text{Carries}(c_1, d, x_1, y_1) \wedge \text{Passes}(f, d, y_1, x_2) \wedge \text{Carries}(c_2, d, x_2, y_2) \\
 & & \qquad\qquad \supset \text{label}(d) \leq \text{clearance}(y_2)] \\
\{4\} & 4. & (\forall a : \text{Component})(\forall y : \text{Input\_Port})[\text{Port\_Of}(y, a) \supset \text{clearance}(y) \leq \text{clearance}(a)] \\
\{5\} & 5. & (\forall s_1, s_2, s_3 : \text{Security\_Label})[s_1 \leq s_2 \wedge s_2 \leq s_3 \supset s_1 \leq s_3] \\
\{1\} & 6. & \text{Flows}(d_0, rm, ap) \\
 & & \qquad \supset \text{Carries}(\text{rm\_to\_filter\_channel}, d_0, \text{rm's\_ar\_port}, \text{filter\_in\_port}(rm)) \\
 & & \qquad\qquad \wedge \text{Passes}(\text{mls\_filter}, d_0, \text{filter\_in\_port}(rm), \text{filter\_out\_port}(rm)) \\
 & & \qquad\qquad \wedge \text{Carries}(\text{filter\_to\_ap\_channel}(rm), d_0, \text{filter\_out\_port}(rm), \text{ap's\_ar\_ports}(rm)) \\
\{3\} & 7. & \text{Carries}(\text{rm\_to\_filter\_channel}, d, \text{rm's\_ar\_port}, \text{filter\_in\_port}(rm) \\
 & & \qquad \wedge \text{Passes}(\text{mls\_filter}, d_0, \text{filter\_in\_port}(rm), \text{filter\_out\_port}(rm)) \\
 & & \qquad \wedge \text{Carries}(\text{filter\_to\_ap\_channel}(rm), d_0, \text{filter\_out\_port}(rm), \text{ap's\_ar\_ports}(rm)) \\
 & & \qquad\qquad \supset \text{label}(d_0) \leq \text{clearance}(\text{ap's\_ar\_ports}(rm)) \\
\{1,3\} & 8. & \text{Flows}(d_0, rm, ap) \supset \text{label}(d_0) \leq \text{clearance}(\text{ap's\_ar\_ports}(rm)) \\
\{4\} & 9. & \text{Port\_Of}(\text{ap's\_ar\_ports}(rm), ap) \supset \text{clearance}(\text{ap's\_ar\_ports}(rm)) \leq \text{clearance}(ap) \\
\{2,4\} & 10. & \text{clearance}(\text{ap's\_ar\_ports}(rm)) \leq \text{clearance}(ap) \\
\{5\} & 11. & \text{label}(d_0) \leq \text{clearance}(\text{ap's\_ar\_ports}(rm)) \wedge \text{clearance}(\text{ap's\_ar\_ports}(rm)) \leq \text{clearance}(ap) \\
 & & \qquad\qquad \supset \text{label}(d_0) \leq \text{clearance}(ap) \\
\{2,4,5\} & 12. & \text{label}(d_0) \leq \text{clearance}(\text{ap's\_ar\_ports}(rm)) \supset \text{label}(d_0) \leq \text{clearance}(ap) \\
\{1,2,3,4,5\} & 13. & \text{Flows}(d_0, rm, ap) \supset \text{label}(d_0) \leq \text{clearance}(ap) \\
\{1,2,3,4,5\} & 14. & (\forall d : \text{Labeled\_Data})[\text{Flows}(d, rm, ap) \supset \text{label}(d) \leq \text{clearance}(ap)]
\end{array}
$$

Figure 4. Transformed formal proof that dataflow from rm to ap satisfies the security policy

3.5 Completing the proof

The image of the first axiom under μ says that every labelled datum that flows from rm to ap is carried to the filter from rm, passed through the filter, and then carried to ap from the filter. Just as in the case of the first axiom, this is a fact about the particular architecture that is either an axiom of the concrete theory, or easily and automatically derivable from axioms of the concrete theory. The mapping μ leaves the second axiom unchanged. This will certainly be an axiom of the concrete theory, as well as the abstract theory. The image of the third axiom is a bit more complex. It states that the combination of the two channels and the filter enforces the security property. It is quite unlikely that this would be among the chosen axioms of the concrete-level theory, since it is the filter alone, effectively, that is enforcing security. Still, it is easy to see that this formula must be a consequence of axioms of the concrete theory: the security model requires that channels that do not enforce security can only connect ports with matching clearances, and one of the defining properties of an MLS component is that it only supplies data at an output port if the classification of the data is less than to equal to the clearance of the port. A formalization of this proof from particular axioms we use in the SDTP security verification is shown in Figure 5.

{1}	1.	$(\forall c : \mathsf{Channel})(\forall d : \mathsf{Labeled\_Data})(\forall x : \mathsf{Output\_Port})(\forall y : \mathsf{Input\_Port})$ $[\mathsf{Carries}(c, d, x, y) \supset \mathsf{clearance}(x) = \mathsf{clearance}(y)]$	Axiom specifying connection constraint imposed by security model
{2}	2.	$(\forall f : \mathsf{MLS\_Component})(\forall d : \mathsf{Labeled\_Data})(\forall y : \mathsf{Input\_Port})(\forall x : \mathsf{Output\_Port})$ $[\mathsf{Passes}(f, d, y, x) \supset \mathsf{label}(d) \leq \mathsf{clearance}(x)]$	Axiom characterizing MLS components
{3}	3.	$(\forall x)(\forall y)(\forall z)[x = y \supset [z \leq x \equiv z \leq y]]$	Instance of identity axiom schema
{1}	4.	$\mathsf{Carries}(c_2, d_0, x_2, y_2) \supset \mathsf{clearance}(x_2) = \mathsf{clearance}(y_2)$	Universal instantiation (1)
{2}	5.	$\mathsf{Passes}(f_0, d_0, y_1, x_2) \supset \mathsf{label}(d_0) \leq \mathsf{clearance}(x_2)$	Universal instantiation (2)
{1, 2}	6.	$\mathsf{Carries}(c_1, d_0, x_1, y_1) \wedge \mathsf{Passes}(f_0, d_0, y_1, x_2) \wedge \mathsf{Carries}(c_2, d_0, x_2, y_2)$ $\supset \mathsf{label}(d_0) \leq \mathsf{clearance}(x_2) \wedge \mathsf{clearance}(x_2) = \mathsf{clearance}(y_2)$	Tautological consequence (3,4)
{3}	7.	$\mathsf{clearance}(x_2) = \mathsf{clearance}(y_2) \supset [\mathsf{label}(d_0) \leq \mathsf{clearance}(x_2) \equiv \mathsf{label}(d_0) \leq \mathsf{clearance}(y_2)$	Universal instantiation (3)
{1, 2, 3}	8.	$\mathsf{Carries}(c_1, d_0, x_1, y_1) \wedge \mathsf{Passes}(f_0, d_0, y_1, x_2) \wedge \mathsf{Carries}(c_2, d_0, x_2, y_2)$ $\supset \mathsf{label}(d_0) \leq \mathsf{clearance}(y_2)$	Tautological consequence (6,7)
{1, 2, 3}	9.	$(\forall c_1 : \mathsf{Channel})(\forall f : \mathsf{MLS\_Component})(\forall c_2 : \mathsf{Channel})(\forall d : \mathsf{Labeled\_Data})$ $(\forall x_1 : \mathsf{Output\_Port})(\forall x_2 : \mathsf{Output\_Port})(\forall y_1 : \mathsf{Input\_Port})(\forall y_2 : \mathsf{Input\_Port})$ $[\mathsf{Carries}(c_1, d, x_1, y_1) \wedge \mathsf{Passes}(f, d, y_1, x_2) \wedge \mathsf{Carries}(c_2, d, x_2, y_2)$ $\supset \mathsf{label}(d) \leq \mathsf{clearance}(y_2)]$	Universal generalization (8)

Figure 5. Proof of image of abstract-level formula 3 under μ from axioms of concrete theory

Discovery of this proof is easy. The form of the desired conclusion—a conjunction of conditions on Carries and Passes in the antecedent, and the comparison of label to clearance in the consequent—immediately suggests the use of the axioms on lines 1 and 2 of the proof. So it should be quite plausible that the proof can be discovered without human intervention by the transformation system. The interpretation mapping U does not affect the images of the remaining two axioms; they remain general axioms in the security model. So, by combining the proof in Figure 5 with the proof in Figure 4, we obtain a proof of the security property from axioms of the concrete theory. Moreover, this proof is recognizably a formalization of our informal argument (Moriconi, Qian, Riemenschneider & Gong 1997, p. 890) that the concrete architecture satisfies the security policy.

4. GENERALIZING FROM THE EXAMPLE

The idea of using the architectural transformation to transform the proof that the more abstract architecture has a desired property into a proof that the more concrete architecture has the property worked well for this rather simple, but real-world, example. Is there any reason to believe that it will work equally well in other cases?

Recall that the standard criterion for correctness of an implementation mapping μ of an abstract logical theory T_1 in a more concrete theory T_2 is that μ must interpret T_1 in T_2, i.e., it must be the case that, for every formula **A** in the language of T_1,

$$T_1 \vdash \mathbf{A} \Rightarrow T_2 \vdash \mu(\mathbf{A})$$

If μ interprets T_1 in T_2, an easy inductive argument shows that μ maps formal proofs from T_1 to formal proofs from $\mu[T_1]$ that can be extended to proofs from T_2. If **A** is an axiom of T_1, then, since μ is a theory interpretation, μ(**A**) is derivable from T_2. Because μ is defined so that connectives pass through it, μ maps tautological consequence steps to tautological consequence steps. Similarly, μ maps universal instantiation and universal generalization steps to universal instantiation and generalization steps, respectively. Thus, μ maps formal proofs from abstract axioms to formal proofs from images of abstract axioms, and images of abstract axioms can always be proved from concrete axioms, as shown in Figure 6.

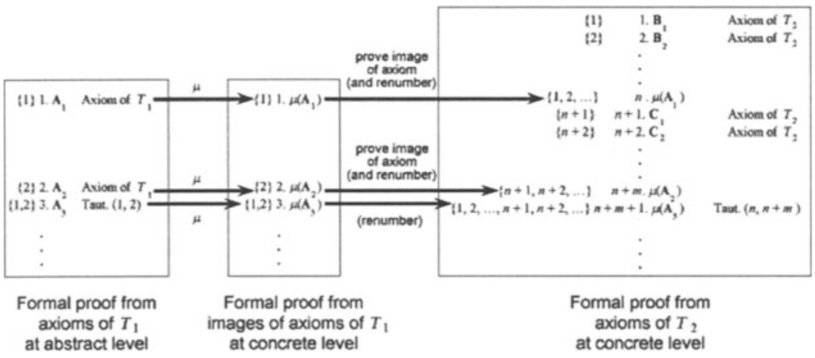

{1} 1. A1 Axiom of T1
{2} 2. A2 Axiom of T1
{1,2} 3. A3 Taut. (1, 2)
μ
μ
μ
{1} 1. μ(A1)
{2} 2. μ(A2)
{1,2} 3. μ(A3)
prove image of axiom (and renumber)
prove image of axiom (and renumber)
(renumber)
{1} 1. B1 Axiom of T2
{2} 2. B2 Axiom of T2
{1, 2, ...} n. μ(A1)
{n+1} n+1. C1 Axiom of T2
{n+2} n+2. C2 Axiom of T2
{n+1, n+2, ...} n+m. μ(A2)
{1, 2, ..., n+1, n+2, ...} n+m+1. μ(A3) Taut. (n, n+m)
Formal proof from axioms of T1 at abstract level
Formal proof from images of axioms of T1 at concrete level
Formal proof from axioms of T2 at concrete level

Figure 6. Interpretation of formal proofs

So, if an architectural transformation step is correct, in the standard sense, the corresponding interpretation mapping will map formal proofs to formal proofs containing gaps that can be filled. A fortiori, an abstract-level formal proof of some particular property of interest—say, satisfaction of a security policy—will be mapped to a proof that the implementation also has (the implementation-level analogue of) the property. Since the replacement of the secure channel from rm to ap by a pair of channels and a filter is evidently correct, it is not surprising that the FIT mapping sends the abstract-level security proof to a concrete-level security proof.

It follows that the proof-carrying architecture approach allows the architect to perform arbitrary correct transformations when implementing an abstract architecture, provided the transformation system that supports the approach is clever enough to find the proofs of images of axioms.

The question remains: In general, how hard is it to discover these proofs? In our experience, it is invariably quite easy, because we deal with refinement patterns that make only small changes in representation of the architecture. Indeed, the example in Figure 5 is representative of the complexity of most of these proofs. At lower levels in the SDTP hierarchy, there are more gaps to be filled in—because lower-level architectural theories are more complex, and proofs are based on a larger number of axioms—but the size of the gaps is about the same. We are confident that considerable automated support for finding proofs to fill the gaps can be provided.

Finally, it should be noted that incorrect transformations that happen to preserve the proof of the property of interest will also be judged acceptable on the proof-carrying architectures approach. Therefore, it is well-suited to the case where the focus is on obtaining an implementation with some particular desirable property — i.e., when a weaker-than-usual correctness criterion is adequate — and placing minimal constraints on the architect's implementation options is preferred, as is the case in SDTP.

5. RELATED WORK

Although there is a large and growing literature on formal software transformation, nearly all of it is oriented toward maintaining functional correctness, rather than system structure. Similarly, there is a large body of literature on architectural refinement and composition, nearly all of it employing semiformal representation and analysis techniques, at best. The comparatively few papers on formal refinement of architectural structure include Broy's work on component refinement (Broy 1992), Brinksma, et al.'s, work on connector refinement (Brinksma, Jonsson & Orava 1991), Philipps and Rumpe's recent work on refinement of information flow architectures (Philipps & Rumpe 1997), and the work described in our own earlier papers. Also closely related is work by Garlan's group (Abowd, Allen, and Garlan 1995), Luckham's group (Luckham, Augustin, Kenney, Vera, Bryan & Mann 1995) , and Moriconi and Qian's work on formally representing the semantics of connectors and relating semantic models at different levels of abstraction (Moriconi & Qian 1994). But, the emphasis in all these cases has always been on verification of general refinement patterns, rather than checking particular steps.

Necula and Lee's work on proof-carrying code and its applications (Necula & Lee 1996, 1997, 1998) introduced the notion of replacing verification by checking in the context of compilation. The work described in this paper can be viewed as generalizing their ideas about code refinement transformations to architectural transformations, both refinements and abstractions.

6. CONCLUSIONS

Transformational development of architectures can guarantee that implementations are correct by restricting the architect to a stock of verified transformations. But such a correctness guarantee is quite brittle, since use of a single non-verified transformation voids it. Moreover, if many transformations are used, and the verification of each is difficult, then confidence in the correctness of the implementation may be less than desired. Checking particular refinement steps offers a way of allowing the architect greater freedom, and of achieving higher levels of confidence that the implemented architecture has the desired properties.

Our initial approach to checking, based on the idea of proof-carrying architectures, is especially well suited to the case where the main requirement is high confidence that the implementation has some specific property. The property is shown to hold at some abstract level, and every refinement is produced by application of a transformation known to preserve the property, or is checked for correctness by making sure that the transformation preserves the proof of the desired property, or both.

The main limitation of this first approach to checking is that properties are checked one at a time. We are exploring other approaches to checking that allow an entire class of properties to be checked at once. One that seems particularly promising is based on the idea of applying the simplified technique for proving implementation mapping correctness (Riemenschneider 1997) to development steps at architecture definition-time. This complementary approach to checking will allow the correctness of steps to be checked, relative to our strong correctness criterion, rather than checking one or a few properties of interest. But it can be applied only to complete architectural descriptions of single structures, not to descriptions of varied families of architectural structures. The proof-checking architectures approach applies equally well to descriptions of single structures and descriptions of families.

As mentioned above, our preliminary experiments with proof-carrying architecture are being performed with the PVS verification system (Owre, Rushby & Shankar 1992). Improved support for working with proof-carrying architectures, including automated discovery of the gap-filling proofs, is being implemented as part of the Xform[4] system, an enhanced version of our present architectural correctness checking toolset. *Xform*, pronounced *transform*, is a recursive acronym for "*Xform, for orderly reification[5] and maintenance*." Xform will support transformational development and maintenance of architectural descriptions written in languages such as SADL (Moriconi & Riemenschneider 1997) and ACME (Garlan, Monroe & Wile 1997).

ACKNOWLEDGEMENTS

This research was supported by the Defense Advanced Projects Research Agency (DARPA) Information Technology Office (ITO) under contracts F30602-95-C-0277 and F30602-97-C0040, whose support is hereby gratefully acknowledged. I would also like to thank Axel van Lamsweerde for his many helpful comments on the first draft of this paper.

REFERENCES

Abowd, G., Allen, R., and Garlan, D., (1995) Formalizing style to understand descriptions of software architecture. Tech. Rep. CMU-CS-95-111, School of Computer Science, Carnegie Mellon University.

Brinksma, E., Jonsson, B., and Orava, F., (1991), Refining interfaces of communicating systems. *Proceedings of* TAPSOFT '91, S. Abramsky and T.S.E. Maibaum, Eds., Springer-Verlag, pp. 71—80

Broy, M. (1992), Compositional refinement of interactive systems. Tech. Rep. No. 89, Digital Systems Research Center, Palo Alto.

Enderton, H. B. (1972), *A Mathematical Introduction to Logic*. Academic Press.

Garlan, D., Allen, R., and Ockerbloom, J. (1994), Exploiting style in architectural design environments. *In Proceedings 2nd* ACM SIGSOFT Symposium on Foundations of Software Engineering SIGSOFT '94, ACM Press, pp. 179—185.

Garlan, D., Monroe, R.~T., and Wile, D. (1997), Acme: An archiectural description interchange language. In *Proceedings of* CASCON '97. Available at

[4]*Xform* is a common mathematical shorthand for *transform*.

[5]*To reify* means to make actual. Thus, reification of software architectures is the process of turning them into actual implementations.

http://www.cs.cmu.edu/afs/cs/project/abel/www/acme-web/v3.0/white-paper-v3.0/white-paper.html.

Hodges, W. (1993), *Model Theory.* Cambridge University Press.

Lemmon, E. J. (1987), *Beginning Logic*, second ed. Chapman and Hall.

Luckham, D. C., Augustin, L. M., Kenney, J. J., Vera, J., Bryan, D., and Mann, W. (1995), Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering 21*, 4, pp. 314—335.

Mates, B. (1972), *Elementary Logic*, second ed. Oxford University Press.

Moriconi, M., and Qian, X. (1994), Correctness and composition of software architectures. *In Proceedings 2nd ACM Symposium on Foundations of Software Engineering (SIGSOFT '94)* , ACM Press, pp. 164—174.

Moriconi, M., Qian, X., and Riemenschneider, R. A. (1995), *Correct architecture refinement.* IEEE Transactions on Software Engineering 21, 4, 356—372. Available at http://www.csl.sri.com/sadl/tse95.ps.gz.

Moriconi, M., Qian, X., Riemenschneider, R. A., and Gong, L. (1997), Secure software architectures. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pp. 84—93. Available at http://www.csl.sri.com/sadl/sp97.ps.gz.

Moriconi, M., and Riemenschneider, R. A. (1997), Introduction to SADL 1.0: A language for specifying software architecture hierarchies. Tech. Rep. SRI-CSL-97-01, Computer Science Laboratory, SRI International. Available at http://www.csl.sri.com/sadl/sadl-intro.ps.gz.

Necula, G. C., and Lee, P. (1998), The design and implementation of a certifying compiler. Submitted to PLDI '98. Available at http://www.cs.cmu.edu/~necula/pldi98.ps.gz.

Necula, G. C., and Lee, P. (1996), Proof-carrying code. Tech. Rep. CMU-CS-96-165, School of Computer Science, Carnegie Mellon University. Available at http://www.cs.cmu.edu/~necula/tr96-165.ps.gz.

Necula, G. C., and Lee, P. (1997) Efficient representation and validation of logical proofs. Tech. Rep. CMU-CS-97-172, School of Computer Science, Carnegie Mellon University,. Available at http://www.cs.cmu.edu/~necula/tr97-172.ps.gz.

Owre, S., Rushby, J. M., and Shankar, N. (1992), PVS: A prototype verification system. In *11th International Conference on Automated Deduction* (CADE) (Saratoga, NY,), D. Kapur, Ed., vol. 607 of Lecture Notes in Artificial Intelligence, Springer-Verlag, pp. 748—752.

Philipps, J., and Rumpe, B. (1997), Refinement of information flow architectures. *In Proceedings of the First IEEE International Conference of Formal Engineering Methods* (ICFEM '97), pp. 203—212. Available at http://www4.informatik.tu-muenchen.de/papers/icfem_rumpe_1997_Publication.html.

Riemenschneider, R. A. (1997), A simplified method for establishing the correctness of architectural refinements. SRI CSL Dependable System Archiecture Group, Working Paper DSA-97-02. Available at http://www.csl.sri.com/sadl/simplified.ps.gz..

Riemenschneider, R. A. (1998), Correct transformation rules for incremental development of architecture hierarchies. SRI CSL Dependable System Archiecture Group, Working Paper DSA-98-01. Available at http://www.csl.sri.com/sadl/incremental.ps.gz.

Shoenfield, J. R. (1967), *Mathematical Logic.* Addison-Wesley.

X/Open Company. (1993), *Distributed Transaction Processing: Reference Model.* Apex Plaza, Forbury Road, Reading, Berkshire RG1 1AX, U.K., November 1993.

Developing Dependable Systems Using Software Architecture

Titos Saridakis & Valérie Issarny
INRIA/IRISA, Campus de Beaulieu, 35042 Rennes Cédex, France
{saridaki, Valerie.Issarny}@irisa.fr

Abstract: The construction of dependable software systems is recognized as a complex task: the system developer has to address the usage of fault tolerance techniques in addition to the design of the functional aspects that are specific to the system. This paper proposes a framework aimed at easing the development of dependable systems by providing software designers with a repository of dependable software architectures. A dependable software architecture shows how to integrate a fault tolerance technique with a given system so as to make the system dependable. Furthermore, the dependability behaviors of architectures are formally specified, which allows to unambiguously interpreting the various fault tolerance techniques as well as to organize the repository of corresponding architectures into a refinement-based lattice structure.

Key words: Dependability, formal specification, software architecture, software reuse, specification refinement.

1. INTRODUCTION

Making a system dependable is recognized as a complex task. In addition to the treatment of functional aspects that are system-specific, the system's designer has to cope with the integration of the fault tolerant mechanisms that satisfy the system's dependability requirements. However, the field of dependability has reached a sufficient level of maturity to capture its various ramifications. In particular, there exist a significant number of fault tolerant mechanisms to handle various dependability needs over different system platforms. Thus, there is an *a priori* knowledge of the mechanisms that are eligible to make a system dependable with respect to the system's dependability requirements and underlying platform. Furthermore, the understanding of fault-tolerance mechanisms and associated abstractions enables a separation of concerns in system design by addressing independently the design regarding functional and dependability aspects. In that context, we propose a framework for making a system dependable through the reuse of appropriate fault tolerance abstractions.

Our work builds on results of the software architecture field (Perry and Wolf 1992, Shaw and Garlan 1996). A system's software architecture abstractly describes the system's gross organization in terms of components (i.e., units of computation) and connectors (i.e., units of interaction). This allows the practical use of formal methods to define the behaviors of components and connectors, and to carry out complementary system analyses. Our framework for the construction of dependable systems consists of characterizing dependable software architectures that are generic with respect to the base functional architectural elements (i.e., functional components and connectors among them). The dependability behaviors of the architectures are further defined formally, which enables their unambiguous interpretation, as well as to organize the set of dependable architectures according to a refinement relation over their behavior. Practically, the developer is provided with a repository of dependable architectural patterns from which he may select the one that meets the dependability requirements of his system. Ultimately, the fault tolerance constituents of a dependable architecture may correspond to implemented mechanisms. Such mechanisms can be directly integrated with the system's functional structure according to the structure shown by the dependable architecture.

This paper is organized as follows. Section 2 details our approach to the formal specification of dependability behaviors. Section 3 introduces our framework for making systems dependable, presenting a repository of dependable software architectures. Finally, we conclude in Section 4, summarizing our contribution and comparing it with related work.

2. FORMAL SPECIFICATION OF DEPENDABILITY BEHAVIOR

To be practically beneficial for software development, a formal framework should satisfy two conditions:

1. It should be easy to understand and use.
2. It should be expressive enough to capture all (or at least a big majority) of the targeted properties (i.e., properties relating to dependability in our case).

Both these conditions are satisfied by an extension of predicate logic with the *precedence* relation (Lamport 1978) (binary operator « $<$ ») specifying a partial order in which predicates are verified. Based on the precedence relation, we define the relations *eventually* (unary operator « $\Diamond$ ») and *in the past* (unary operator « ∇ ») which denote that a predicate will be verified in the future or was verified in the past. The extended predicate logic provides comprehensible and easy to employ means for combining the constraints on system states that should be reached after a failure with the partial order of actions that should be performed to reach these states. Notice that the use of temporal logic relations is not indispensable for modeling the temporal precedence of the predicate. Indeed, means have been invented like *history lists*, which are employed by a number of approaches (e.g., see (Chrysanthis and Ramamritham 1994) and (Stoller and Schneider 1996)) in order to avoid temporal relations for ordering the occurrences of events in a system and to remain purely first order logic. However, we use them because we believe that they render the formulas more legible. The formal framework we use is presented hereafter, followed by our approach to the specification of system behaviors with respect to dependability, introducing the specification of dependability properties and a refinement relation over them.

2.1 Formal framework

A *system* is a set of variables, which can be assigned different values according to the system specifications. A *state* of the system is a mapping of variables to values, where the values of some variables can be undefined. When the values of one or more variables lay outside the range defined in system's specifications, a *failure* is said to occur. An *execution* of a system is a partially ordered set of system states, where one state in the set is distinguished as being the initial state (i.e., the state preceding all other states in the set). An *object*[1] of the system is an entity having some state. Hence, a system can be seen as a set of objects. An *action* is a state transition, which can be caused by some internal object computations, or by some I/O operation. Actions are associated with objects and we assume deterministic actions, i.e., given the specifications of an object, an object state and an action on that state, the resulting state after performing the action is uniquely defined. However, we do not constrain the choice of the next action to be performed, which can be a random choice among different alternatives. Hence, although actions are deterministic, the execution of an object and, consequently, the execution of the entire system are non-deterministic. In

[1] The term object signifies a logical entity and not entities specific to programming languages (e.g. C++ objects).

this context, an *event* is the execution of some actions or the reach of some state. In the remainder of this document, we use the following notations:

- Objects are denoted by the first five lower-case Greek letters, primed or followed by a subscript value (e.g., α , β_i, *etc*).
- σ, primed or followed by a subscript value, denotes a system state. For object states, we prefix the object name (e.g., $\alpha.\sigma$). We neglect the object name when it is obvious in a given context.
- Σ denotes system specifications. For object specifications, we prefix the object name (e.g., $\alpha.\Sigma$).
- X denotes a system execution which is a partially order set of system states. When followed by the superscript ^{C}, it denotes a failure-free execution.
- Actions are written in lower-case italics followed by a list of arguments in parentheses. To distinguish among actions of different objects, we prefix the object name to the action (e.g., α.*import*(β, data)). We neglect the object name when it is obvious in a given context.

The structural elements of the system model presented above do not suffice to describe the properties of a specific system (i.e., the relations among constituent objects, their interactions, their invariants, and their constraints). For this, a set of predicates is needed to capture the essential properties of system entities. This set of predicates should be minimal in order to be easy to use and understand. We present below a set of predicates that capture the fact that the system is in a given state, and the execution of I/O actions. Notice that this set of predicates is not unique; another set of predicates, richer or more frugal, can be chosen if it facilitates the system programmer's reasoning (e.g., additional predicates that can be defined are *init, exit, begin, commit*, and *abort,* to describe the actions related to object initialization and termination, or the actions related to transactional properties). In the remainder of this document, we use the following predicates:

- The predicate expressing that a system is in state σ is introduced by the unary operator [ ], i.e., $[\sigma]$ is true when the system is in state σ . Similarly $[\alpha.\sigma]$ is verified when object α is in state σ .
- The predicate *export* expressing the I/O action performed by an object α when it sends to an object β the data d. The data d sent, are a function of some α 's state preceding the I/O action, and if some β 's state is a function of data d, the export action precedes this state. More formally, the predicate is defined as:
  export$(\alpha, \beta, d) \equiv (\exists \alpha.\sigma : \nabla[\alpha.\sigma] \wedge d = f(\alpha.\sigma)) \wedge (\beta.\sigma = g(d) \Rightarrow \Diamond[\beta.\sigma])$
- The predicate *import* expressing the I/O action performed by an object β when it receives the data d sent by object α. The export of data d

from α to β preceded, and some state of β after receiving d is a function of the received data. More formally, the predicate is defined as:
$\mathit{import}(\alpha, \beta, d) \equiv \nabla \mathit{export}(\alpha, \beta, d) \wedge (\exists \beta.\sigma : \beta.\sigma = g(d) \wedge \Diamond[\beta.\sigma])$

Notice that the predicates *export* and *import* correspond to the $\alpha.\mathit{export}(\beta,d)$ and $\beta.\mathit{import}(\alpha,d)$ actions respectively.

2.2 Dependability properties

Given the above formal framework, we are able to define dependability properties, which serve to characterize dependability behaviors of software architectures. Let us point out here that the dependability properties introduced in the following, reflect more the authors' perspective on the issue rather than a widely acknowledged characterization. Alternative specifications of dependability behaviors can be envisioned. In the same way, there may exist alternative interpretations for the terms we use to qualify the dependability properties. Here, we base our work on the fault tolerance perspective introduced by Laprie (Laprie 1992).

The important point we want to make with respect to our approach to the specification of dependability properties is that it enables us to characterize the various behaviors of a system in the presence of failure, which are attainable using existing fault tolerance techniques. The set of these behaviors may further be expanded as new fault tolerance techniques emerge. In the remainder, we present the specification of some representative dependability properties so as to give the reader, the intuition of how dependability properties are characterized in general. Basically, dependability properties fall into two groups:

1. *Abstract properties* specified in terms of system states, which are defined independently of any fault tolerance technique. They serve to characterize the dependability behavior of an overall architecture, when this behavior is too abstract to associate a specific fault tolerance technique with it.
2. *Concrete properties* specified in terms of system actions, whose definition is closely related to some fault tolerance technique. They serve to characterize the dependability behaviors associated with architectural elements, with respect to a given fault tolerance technique.

Let us first give abstract properties defined at the state level. The most abstract dependability property, simply qualified as *Dependability*, ensures that a system makes progress despite the occurrence of failure. The *Safety* property defines that, after a failure, the system should enter an error-free state, which is some subset of a state, reached before the occurrence of the failure. The basic characteristic of this abstraction is the removal of the failure products. The specification of the *Availability* property indicates that the state reached after a failure is a state contained in some failure-free system execution. The basic characteristic of this abstraction is the repair of failure effects. Another specification is the *Reliability* property, which defines that the state reached after a failure includes a state that should have been reached in the absence of failures. The basic characteristic of this property is the transition to the expected state despite the occurrence of failures. More specific dependability properties are the ones of *Detection* and *Fmask* where the former characterizes failure detection and the latter the system capability to mask the occurrence of failures. Let *faulty* be the predicate expressing that a system state contains an erroneous mapping of variables to values, i.e., $\textit{faulty}(\sigma)$ is true when some of the variables of σ have been assigned values not defined by system's specifications. Similarly $\textit{faulty}(\alpha.\sigma)$ is verified when the object state $\alpha.\sigma$ contains an erroneous mapping from variables to values.

Table 1: The formal specifications of some dependability properties.

$\text{Dependability}(S) \equiv ([\sigma] \wedge \textit{faulty}(\sigma)) \Rightarrow \exists\, \sigma' \in \Sigma : [\sigma] < [\sigma']$
$\text{Safety}(S) \equiv ([\sigma] \wedge \textit{faulty}(\sigma)) \Rightarrow (\exists\, \sigma',\sigma'' \in \Sigma : ([\sigma] < [\sigma']) \wedge ([\sigma''] < [\sigma]) \wedge (\sigma' \subseteq \sigma''))$
$\text{Availability}(S) \equiv ([\sigma] \wedge \textit{faulty}(\sigma)) \Rightarrow (\exists\, \sigma' \in \Sigma : ([\sigma] < [\sigma']) \wedge (\sigma' \in X^C))$
$\text{Reliability}(S) \equiv ([\sigma] \wedge \textit{faulty}(\sigma)) \Rightarrow (\exists\, \sigma',\sigma'' \in \Sigma : ([\sigma] < [\sigma']) \wedge (\sigma'' \in X^C) \wedge (\forall\, \sigma_p : [\sigma_p] < [\sigma] \Rightarrow [\sigma_p] < [\sigma'']) \wedge (\sigma'' \subseteq \sigma'))$
$\text{Detection}(S) \equiv [\sigma] \wedge \textit{faulty}(\sigma)$
$\text{Fmask}(S) \equiv \forall\, \sigma \in \Sigma : (\exists\, \sigma_f : \textit{faulty}(\sigma_f) \wedge ([\sigma] < [\sigma_f])) \Rightarrow (\exists\, \sigma' \in \Sigma : ([\sigma_f] < [\sigma']) \wedge (\sigma \subseteq \sigma'))$
$\text{DetectionObj}(\alpha) \equiv (\exists\, \beta : [\beta.\sigma] \wedge \textit{faulty}(\beta.\sigma)) \Rightarrow [\beta.\sigma] < \textit{export}(\alpha, \varepsilon, f(\beta\text{-failed}))$
$\text{FmaskObj}(\alpha) \equiv ([\alpha.\sigma] \wedge \textit{faulty}(\alpha.\sigma)) \Rightarrow (\exists\, \beta : \beta.\Sigma \equiv_s \alpha.\Sigma \wedge [\alpha.\sigma] < [\beta.\sigma] \wedge \neg\textit{faulty}(\beta.\sigma)) \wedge \exists \alpha.\sigma' : (([\alpha.\sigma'] < [\alpha.\sigma]) \Rightarrow (\exists\, \beta.\sigma' : (\beta.\sigma' = \alpha.\sigma' \wedge [\beta.\sigma'] < [\beta.\sigma]))))$

The upper part of Table 1 gives the specifications of the aforementioned dependability properties, for a system S. The properties in the upper part of Table 1 characterize only the system state that is reached after a failure occurrence. They do not make explicit the system objects that are involved in fault treatment nor the needed interactions among them. This is captured by concrete properties, defined at the action level. For instance, the *Detection* and *Fmask* properties may be respectively revised into the specification of *DetectionObj* and *FmaskObj*. The specification of the former expresses the fact that a system object transmits a message to some other object in the system, after a failure occurred. This message contains the information of the occurred failure, which implies that the transmitting object captures this knowledge in its state. Similarly, the specification of the latter expresses the fact that for a failed object, there exists an equivalent object (not necessarily a different one) which reaches a correct state that follows all the failed object's states preceding the failure. In other words, this means that the state that would have been reached by a given object in the absence of failures, is eventually reached even if a failure occurs on the object in question.

The formal expressions that describe the aforementioned properties are given in the lower part of Table 1. Notice that the interaction events are expressed by the *export* and *import* predicates, and their parameters define the exact interaction pattern between the two objects indicated by the predicate parameters. Object ε is used to signify any object of the environment. In addition, the equivalence of object specifications, noted $\equiv_S$, is defined with respect to the observable behavior of objects, i.e., the specifications of two objects are equivalent if the sequences of *import()* and *export()* actions performed by the objects are equivalent.

As more concrete examples, let us consider the enforcement of dependability for an object, using a replication technique. Achieving replication consists of replicating an object into a group of objects and making the group behave as a single object from the perspective of the group's environment. The behavior of the objects group may differ depending on the replication technique (i.e., active, semi-active, passive) that is used. The formulas of Table 2 characterize the dependability properties for the active and passive replication techniques, where *id(d)* uniquely identifies the data *d* among all the data exchanged in the system. The *id* function is defined so that *id(d) = id(d')* if *d* and *d'* are exported by objects having equivalent specifications and the export actions correspond in the sequences of the I/O actions performed by the objects.

2.3 Refinement relation

Based cn the proposed approach to the specification of dependability properties, we are able to define a refinement relation over these properties. This relation allows refining an initial dependability requirement into more

Table 2: Formal specification of active and passive replication

$\mathrm{Active}(\alpha, N) \equiv \exists\, \alpha_1, \ldots, \alpha_{N-1} : G = \{\alpha, \alpha_1, \ldots, \alpha_{N-1}\} \wedge \mathrm{Replication}(G) \wedge \mathrm{Filter}(G) \wedge \mathrm{AtomicDelivery}(G)$
$\mathrm{Replication}(G = \{\alpha_i\}_{i=1..N}) \equiv \forall\, \alpha_i, \alpha_j \in G : ((\alpha_i.\Sigma \equiv_s \alpha_j.\Sigma) \wedge \neg(\mathit{faulty}(\alpha_i.\Sigma) \Rightarrow \mathit{faulty}(\alpha_j.\Sigma)))$
$\mathrm{Filter}(G = \{\alpha_i\}_{i=1..N}) \equiv \exists\, \beta : (((\alpha_i, \alpha_j \in G) \wedge \mathit{import}(\alpha_i, \beta, d_i) \wedge \mathit{import}(\alpha_j, \beta, d_j) \wedge (\mathrm{id}(d_i) = \mathrm{id}(d_j))) \Rightarrow (\exists!\, \mathit{export}(\beta, \varepsilon, d_0) \wedge (\mathrm{id}(d_0) = \mathrm{id}(d_i) = \mathrm{id}(d_j))))$
$\mathrm{AtomicDelivery}(G = \{\alpha_i\}_{i=1..N}) \equiv (\exists\, \alpha \in G : \mathit{import}(\varepsilon, \alpha, d) \Rightarrow (\forall\, \alpha_i \in G : \mathit{import}(\varepsilon,\alpha_i, d))) \wedge (\exists\, \alpha \in G : (\mathit{import}(\varepsilon, \alpha, d_1) < \mathit{import}(\varepsilon, \alpha, d_2)) \Rightarrow (\forall\, \alpha_i \in G : (\mathit{import}(\varepsilon, \alpha_i, d_1) < \mathit{import}(\varepsilon, \alpha_i, d_2))))$
$\mathrm{Passive}(\alpha) \equiv \exists\, \gamma, \beta : \mathrm{Replication}(\{\alpha, \beta\}) \wedge \mathrm{StableStorage}(\alpha, \gamma) \wedge \mathrm{Restore}(\alpha, \beta , \gamma)$
$\mathrm{StableStorage}(\alpha, \gamma) \equiv \mathit{import}(\alpha, \gamma, f) \wedge (f = \alpha.\sigma) \wedge ([\alpha.\sigma] < \mathit{export}(\alpha, \gamma, f)) \wedge \neg(\exists\, \gamma.\sigma : \mathit{faulty}(\gamma.\sigma))$
$\mathrm{Restore}(\alpha, \beta, \gamma) \equiv (\exists\, \alpha.\sigma' : (([\alpha.\sigma'] < [\alpha.\sigma]) \wedge \mathit{import}(\gamma, \beta, f) \wedge (f = \alpha.\sigma'))) \wedge (\forall\, \varepsilon, d : (([\alpha.\sigma] < \mathit{export}(\varepsilon, \alpha, d)) \Rightarrow \mathit{import}(\varepsilon, \beta, d)))$

concrete dependability properties, which ultimately correspond to the behavior of fault tolerance mechanisms for which an implementation is available. Considering two dependability properties $P_1(S)$ and $P_2(S)$, the latter is a refinement of the former if $P_2(S) \Rightarrow P_1(S)$. For illustration, Figure 1 depicts the refinement relation that holds over the dependability properties introduced in the previous subsection. In the figure, each property P is represented by a box that contains a set of boxes to denote alternative correct refinements of P, and each of these sub-boxes points towards a set of properties whose conjunction is a correct refinement of P.

3. REPOSITORY OF DEPENDABLE SOFTWARE ARCHITECTURES

The proposed specification of dependability properties provides means to unambiguously describe the dependability behavior of an architecture, but it is of limited help from the standpoint of easing the development of dependable systems. To facilitate their use, we propose to attach to each dependability property, the structure (i.e., the software architecture) of the corresponding system with respect to the fault tolerance technique that is used to enforce the given property.

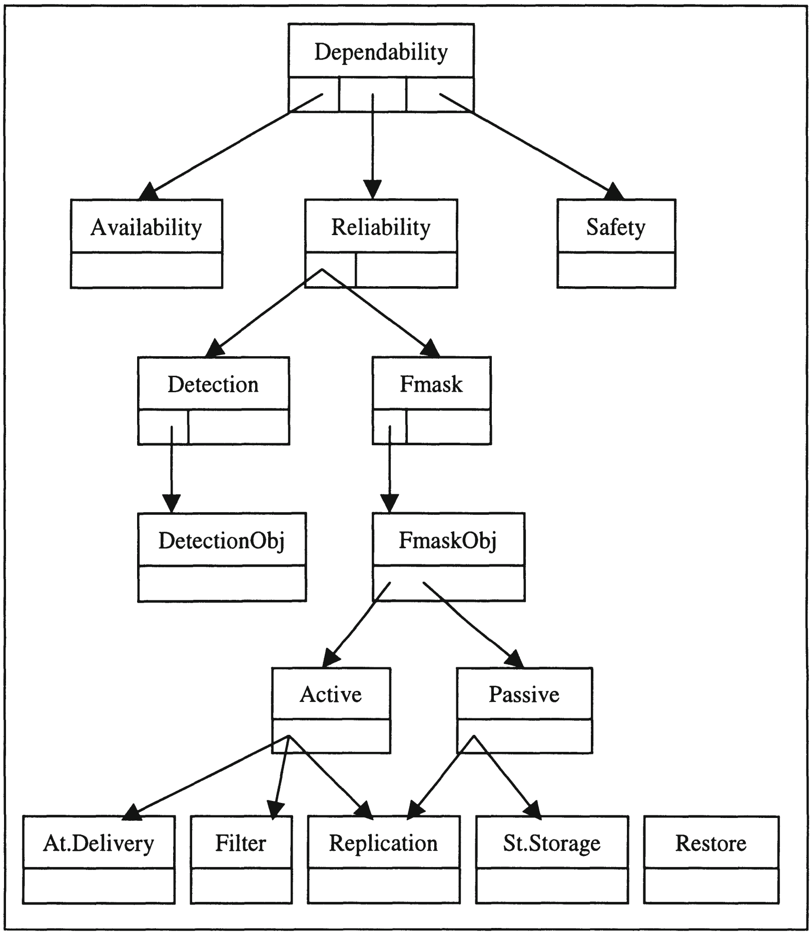


Figure1: Some refinements of the dependability property

The refinement relation over dependability properties provides the adequate base ground to organize the repository of dependable software architectures. The repository is organized as a lattice structure defined according to the refinement relation, and each node stores the acquired knowledge about a given dependability property. For some property *P*, this knowledge includes: (i) the property name, (ii) the formal specification of the dependability property, (iii) the set of dependability properties (through references to adequate nodes) into which P may be refined, and (iv) the dependable software architecture A_P, associated with P.

The repository may be depicted in a way similar to the graph given in Figure 1 except that each node now embeds the description of the *dependable software architecture* corresponding to the property defined by the node. The dependable architecture corresponding to an abstract property is a black-box component embedding the system since the property is too abstract to have a fault tolerance technique associated with it. On the other hand, the architecture defined for a concrete property exposes the system's structure with respect to some fault tolerance technique. The following subsection further elaborates on the description of dependable architectures, which, as shown in Subsection 3.2, may be derived from the specification of dependability properties. Subsections 3.3 and 3.4 then introduce the main functions used for the management of the architecture repository; they relate to the introduction and retrieval of a dependable architecture with respect to a given property.

Prior to detailing the description of dependable software architectures, let us note that we concentrate here on the definition of architectures with respect to the fault tolerance technique that is used to enforce a given dependability property. The proposed architectural description may be enriched when there is an available mechanism to implement the embedded fault tolerance technique. For instance, the architectural definition could then include the specification of the component's interaction protocol (e.g., using Wright (Allen and Garlan 1997)) and of the component's functional interface. In the same way, the definition of connectors could be introduced so as to detail the interaction protocol used by the mechanism. In general, the description of a dependable software architecture includes at least the specification of the dependability behavior of its components, and may be extended using the capabilities of existing ADLs (Architecture Description Languages). In particular, a dependable architecture may be defined using ACME (Garlan *et al.* 1997) so as to exploit different ADLs and thus allow various architecture analyses.

3.1 Dependable software architecture

To be helpful to system developers, the description of dependable architectures must make clear how to compose a dependable system from a base system. The components of a dependable architecture may be of either of the two following kinds: *Generic*, in which case the component corresponds to the initial system that is to be made dependable, or *Dependable*, in which case the component is specifically introduced for enforcing some dependability behavior. Then, given a software architecture providing some concrete dependability property, a system can be integrated with the corresponding fault tolerance technique by mapping the system onto the generic components. We propose the following description for dependable architectures

Dependable Architecture: *Name* =
  Dependability:
    -- Architecture's dependability property --
  Components:
    {*Component Name: TypeComp:* -- Dependability behavior --} +
  Configuration:
    -- Description of a configuration through bindings among components --

where the specifications of dependability behaviors and properties are expressed according to our approach discussed in the previous section.

Table 3: Architectural descriptions associated with the Replication, Filter, and AtomicDelivery properties

Dependable Architecture : Replication =
  Dependability : Replication(G);
  Components : G[i: 1..N] : **Generic:** Replication(G);
  Configuration : *nil* ;

Dependable Architecture : AtomicDelivery =
  Dependability : AtomicDelivery(G);
  Components : G[i: 1..N]: **Generic :** (i:1..N) :
    $(\mathit{import}(\varepsilon, G(i), d) \Rightarrow \forall j \in [1, N] : \mathit{import}(\varepsilon, G(j), d))$
    $\wedge ((\mathit{import}(\varepsilon, G(i), d) < \mathit{import}(\varepsilon, G(i), d')) \Rightarrow$
    $(\forall j \in [1, N] : \mathit{import}(\varepsilon, G(j), d) < \mathit{import}(\varepsilon, G(j), d'))$;
  Configuration : (i: 1..N) : AtomicDelivery.**Import to** G(i).**Import**;

Dependable Architecture : Filter =
  Dependability : Filter(G);
  Components : G[i: 1..N] : **Generic :** *TRUE* ;
    F : Dependable : $(i, j \in [1, N] \wedge \mathit{import}(G(i), F, d) \wedge$
    $\mathit{import}(G(j), F, d') \wedge (id(d) = id(d'))) \Rightarrow$
    $(\exists ! \, \mathit{export}(F, \varepsilon, d'') : (id(d'') = id(d)))$;
  Configuration : (i: 1..N) : G(i).**Export to** F.**Import**;
    F.**Export to** Filter.**Export**;

A dependability behavior may simply be *TRUE* if there is no dependability requirement associated with the architectural element. The type of a component identifies whether the component is generic or dependable. We further assume that each architectural component (including the architecture itself) has an **Import** and an **Export** port. For illustration, Table 3 gives the descriptions of the architectures associated with the *Replication*, *Filter*, and *AtomicDelivery* properties. Let us remark that the proposed architectural descriptions expose only structural information regarding fault tolerance. In particular, only bindings dedicated to fault tolerance are characterized.

Considering the proposed description of dependable architectures, a system *S* may be modified so as to enforce a given dependability property *P* by mapping *S* onto each generic component of the architecture associated with *P* while ensuring the declared dependability behavior, and providing an adequate implementation for the dependability-specific components. Alternatively, the repository of dependable architectures may further be exploited to find out more refined architectures, which possibly correspond to available fault tolerance mechanisms.

3.2 Deriving dependable architectures from properties specifications

Ideally, one would like to have a systematic way to derive the structure of a dependable architecture from its associated formal specification. Although not direct, the proposed specification of dependability properties embeds the needed information. Let us take a close look at dependability properties. From a property specification, we are able to infer:

1. the objects involved in the enforcement of the property, which are all the objects appearing in the specification
2. the objects' behaviors with respect to dependability, which are given by part of the specification that refers to the object
3. the needed interactions among objects, which are given by part of the specification expressed in terms of *import* and *export* predicates.

To systematically infer the above information and hence a dependable architecture, from a property specification, we propose to structure the specification of dependability properties accordingly. For *ObjectType* stating whether the object is generic or not, and parameters *VarName* being of type integer, Table 4 gives the form of the specifications of a property *P*, followed by an illustration of its employment using as an example the *Filter* property.

Table 4. The form of property specification and an example

P(**objects** {*ObjectName : ObjectType*}*; **Ind** {*VarName*}+) ≡
  objects : {*ObjectName : ObjectType* ;}+
  behaviors : {*ObjectName :* -- formula -- ;}+
  configuration : -- formula -- ;

Filter(G : **Generic**[1, N]) ≡
  objects : F : **Dependable** ;
  behaviors : (i: 1..N) : G(i) : *TRUE* ;
    F : $(import(G(i), F, d) \wedge import(G(j), F, d') \wedge id(d) = id(d')) \Rightarrow$
      $(\exists !\ export(F, \varepsilon, d'') : (id(d'') = id(d)))$;
  configuration : $((i: 1..N) : import(G(i), F, d)) \wedge export(F, \varepsilon, d')$;

Intuitively, we can infer from the specification of the *Filter* property that the corresponding dependable architecture is made of the set of generic components G(i) and of the dependable component F. In addition, the formula given in the **configuration** part enables to deduce interaction among components based on the semantics of the *import* and *export* predicates: *import*(α, β, d) as well as *export*(α, β, d) implies that the **Export** port of α is bound to the **Import** port of β. We further recall that ε is used to signify any object of the environment. Thus, *import*(α, ε, d) (resp. *import*(ε, α, d)) signifies that the **Export** (resp. **Import**) port of α is bound to the architecture's **Import** (resp. **Export**) port. The same applies for the *export* predicate. Precisely, the inference of the logical formula and of the software architecture corresponding to a given dependability property is achieved as follows. Let P be defined as:

P (**objects** O_i, $1 \le i \le n$; **var** ...) ≡
  objects : O'_i, $1 \le i \le n'$;
  behaviors : $O_i : B_i$, $1 \le i \le m$;
  configuration : B ;

The corresponding logical formula is equivalent to: $\exists O_1, \ldots, O_n, \exists O'_1, \ldots, O'_{n'} : (B \wedge (\wedge_{i=1..m} B_i))$

Let us remark here that the proposed specification of properties may lead to extend the original specifications. This is exemplified by the new definition of Filter , which extends the original one with the formula stated in the configuration part. As another example, let us consider the *AtomicDelivery* property. The embedded formula $\exists \alpha \in G : import(\varepsilon, \alpha, d) \Rightarrow (\forall \alpha_i \in G : import(\varepsilon, \alpha_i, d))$ relates to the behavior of the αs. It also relates to the architecture's configuration: all the αs are accessible by objects of the environment. Thus, this formula must appear in two parts of the property specification. However, the formula for configuration is simplified into $\forall\ \alpha_i \in G$: *import*(ε, α_i , d). In general, we do not see the required modification of property specification as a major drawback given the resulting benefit for the production of architectural descriptions.

Let us now examine the inference of the architecture associated with P(). It consists of defining the interpretation of each constituent of the property specification in terms of architectural description. The treatment of the **objects** and **behaviors** parts of the specification is direct: each object given in the **objects** lists translates into an architectural component whose type (i.e., dependable or generic) is the one declared in the embedding list; and each object behavior given in **behaviors** is attached to the corresponding architectural component. The interpretation of the **configuration** part is less direct, it requires to interpret each element of the corresponding logical formula. Precisely, a formula defining a configuration is of the form: $\wedge_i P_i$ where each P_i is expressed as either an *import* or an *export* predicate, whose parameters may possibly be universally quantified. Thus, each P_i is translated into bindings among components according to the parameters of the *import* or *export* predicates.

3.3 Updating the repository

Updating the architecture repository requires providing functions for the addition and removal of dependability properties. However, since the treatment of the latter is quite straightforward, we address only the former in the following. The introduction of a dependability property *P* leads to insert the corresponding node *N* within the repository, according to the refinement relation over properties.

Inserting a property: Let us use the following notations:

- P denotes the set of dependability properties.
- N denotes the set of nodes of the repository.
- *Prop(N)* is the function that returns the property defined by node *N*.
- $Anc_N(P)$ denotes the set of immediate ancestor nodes of *N*, with respect to the dependability property *P*.
- Dec_N denotes the set of immediate successor nodes of *N*.
- $POW(X)$ denotes the power-set of *X*.

Let us first consider the introduction of a property *P* refining a property of the repository (i.e., *P* needs not to be conjuncted with another property). For instance, if we consider Figure 1, *P* may be *Reliability* but not *Fmask*, which has to be conjuncted with *Detection* to be a refinement of an existing property. Given our assumption, the node *N* for property *P* must be introduced within the repository in a way that guarantees the following two conditions:

$$\mathbf{C1}(P, Anc_N(P)) = \forall N' \in Anc_N(P) : (P \Rightarrow Prop(N')) \wedge \neg(\exists N'' \in \mathsf{N}\text{-}\{N'\} : (P \Rightarrow Prop(N'') \Rightarrow Prop(N))$$

$$\mathbf{C2}(P, Dec_N) = \forall N' \in Dec_N : ((Prop(N') \Rightarrow P) \wedge \neg(\exists N'' \in \mathsf{N} - \{N'\} : (Prop(N') \Rightarrow Prop(N'') \Rightarrow P)))$$

Let us now consider the introduction of a property P that refines an existing one, when conjuncted with a set of complementary properties. We require all these properties to be inserted in the repository at once, using the following *Insert* function. Given a set of properties $\{P_i\}_{i=1..n}$ to insert and the current nodes of the repository, the function returns the ancestor nodes that are common to all the N_is defining the P_is, with respect to the property $\wedge_{i=1..n} P_i$, and the set of successor nodes for each N_i :

$$Insert : \mathit{POW}(\mathsf{P}) \times \mathit{POW}(\mathsf{N}) \rightarrow \mathsf{P} \times \mathit{POW}(\mathit{POW}(\mathsf{P}))$$

$$\begin{aligned} Insert(\{P_i\}_{i=1..n}, \mathsf{N}) \;=\; & (\cap_{i=1..n}Anc_{Ni}(\wedge_{j=1..n} P_j), \{Dec_{Ni}\}_{i=1..n})) \textbf{ if} \\ & \mathbf{C1}(\wedge_{i=1..n} P_i, \cap_{i=1..n}Anc_{Ni}(\wedge_{j=1..n} P_j)) \textbf{ and} \\ & \forall i \in [1, n] : \mathbf{C2}(P_i, Dec_{Ni}) \end{aligned}$$

When a node defining a concrete property P is created within the repository, the node should be completed with its corresponding architecture description. This is realized by inferring the architecture description from the property specification as discussed in the previous subsection.

Correct architecture refinement: Up to this point, we have seen that the introduction of a property within the repository is achieved according to the refinement relation over dependability properties. Let us consider two properties P_1 and P_2 such that P_2 refines P_1. From the developer's standpoint, this means that the architecture A_2 associated with P_2, may be safely used to enforce property P_1. Let us now assume that the architecture A_1 associated with P_1 was originally selected to make a system dependable, but was later replaced by A_2 (e.g., such a replacement may be due to the availability of the mechanisms embedded by A_2). The replacement of A_1 by A_2 is practical only if both architectures have compatible structures, i.e., A_2 exposes the structure of A_1's architectural elements. In this way, the later replacement of a dependable architecture by an architecture enforcing a stronger property does not impact on the design decision made so far. Thus, when a property P_2 refines a property P_1, we require the architecture A_2

associated with P_2, to be compatible with the architecture A_1 associated with P_1. We say that A_2 *is a correct refinement of* A_1 (with respect to their architectural structures). Let us notice that in the case of architectures corresponding to available mechanisms, the refinement relation over architectures could additionally be constrained according to the definition of (Moriconi *et al.* 1995).

Let us first consider the simplest case that is when P_2 corresponds to a single node: the corresponding architecture A_2 is a correct refinement of an architecture A_1, if P_2 refines the dependable property associated with A_1, and if A_2 defines a set of sub-architectures that maps onto the components of A_1. Let us use the following notations:

- An architecture A is defined by the triplet $(P_A, \mathsf{C}_A, \mathsf{B}_A)$.
- P_A denotes the dependability property of A.
- C_A denotes the components of A.
- $\mathsf{B}_A = \{(C_i, C_i')\}_{i=1..n}, C_i, C_i' \in \mathsf{C}_A$, defines the architectural bindings among A's components.
- $Comp : \mathit{POW}(\mathsf{B}) \rightarrow \mathit{POW}(\mathsf{C})$ is the function that returns the set of components embedded in a given set of bindings.
- A denotes the set of dependable architectures.
- $Beh : \mathsf{P} \times \mathsf{C} \rightarrow \mathsf{P}$ is the function that returns the dependable behavior of a given component belonging to the specification of a given dependability property.

We introduce the following function to identify whether an architecture A_R is a correct refinement of an architecture A, with respect to the architectures' structures:

$$Refine : \mathsf{A} \times \mathsf{A} \rightarrow BOOL$$

$Refine(A, A_R)$ = $\exists$ total function $M : \mathsf{C}_A \rightarrow \mathit{POW}(\mathsf{B}_{AR})$ such that M is *1-to-1* and *onto*, **and**
$\forall\, C, C' \in \mathsf{C}_A : (C \neq C' \wedge Comp(M(C)) \cap Comp(M(C')) = \varnothing$, **and**
$\forall\, C \in \mathsf{C}_A : Dependability(P_{AR}, M(C)) \Rightarrow Beh(P_A, C)$

Dependability gives the dependability behavior of the sub-architecture given by a set of bindings among components:

$$Dependability : \mathsf{P} \times \mathit{POW}(\mathsf{B}) \rightarrow \mathsf{P}$$

$$Dependability(P, \mathsf{B}) = (\wedge_{C_i \in Comp(\mathsf{B})} Beh(P, C_i)) \wedge \wedge_{\forall (C, C') \in \mathsf{B}}(import(C, C', \mathrm{d}) \Rightarrow export(C, C', \mathrm{d}))$$

Let us now consider the case where a conjunction of dependability properties P_i, $1 \leq i \leq N$, is introduced as a refinement of an existing property P. We must define the software architecture A that results from the combination of the set of architectures A_i, $1 \leq i \leq N$, associated with each property P_i, and then verify that A is a correct refinement of the architecture associated with P, according to the definition of *Refine*. We have seen that the components of an architecture subdivide into *generic* and *dependable* components. Let us further recall that generic components correspond to the same functional component that is the software system to be made dependable. Henceforth, the generic components of the A_is correspond to the same components. Thus, generic components are mapped onto the same components in the architecture A, and their dependable behavior is the conjunction of the behavior declared in each of the A_is for generic components.

On the other hand, the dependable components of an architecture are in general specific to this architecture. Thus, the dependable components of A are the union of the dependable components of the A_i. However, there are two cases where dependable components of distinct architectures may have to be merged into a single component. One of these cases is exemplified by the architectures used to enforce *Passive* replication: the γ object is shared by the architectures enforcing *StableStorage* and *Restore*. In general, this case is detected through the definition of the conjunction of properties, which may explicitly share objects. The other situation where dependable components of distinct architectures may be merged is when there is a relation of logical implication between each pair of associated dependable behaviors. Here, we can keep only the dependable component that enforce the strongest dependability behavior among the set of components. So far, we have stated how to infer the set of generic and dependable components of an architecture resulting from the composition of some architectures. The set of bindings among these components are further the ones that are specified for the corresponding components within the A_is.

3.4 Using the repository

Using the architecture repository for the construction of a dependable system consists of retrieving the software architecture associated with the dependability property that is targeted for the system. Let $\perp$ be the undefined node. The retrieval function is defined as *Retrieve* : $\mathsf{P} \rightarrow \mathsf{N} \cup \perp$ with:

$$\begin{aligned} \text{Retrieve(P)} \;=\; & N \;\; \textbf{if}\ (N \in \mathsf{N}) \wedge (Prop(N) \Rightarrow P) \wedge \neg(\exists\, N' \in \mathsf{N} : \\ & \qquad (Prop(N) \Rightarrow Prop(N') \Rightarrow P)),\ \textbf{or} \\ & \perp \;\; \textbf{if}\ \neg(\exists\, N \in \mathsf{N} : (Prop(N) \Rightarrow P)) \end{aligned}$$

The node N returned by the *Retrieve* function allows us to identify all the dependable architectures that are eligible to make a system dependable with respect to the given dependability property. These architectures are all the architectures defined by the nodes of the sub-lattice whose root is *N*. Some of the eligible architectures may possibly be combinations of architectures when properties of the sub-lattice are refined into a conjunction of properties. Architecture combination is achieved according to the approach discussed in the previous subsection. Given eligible architectures, it is up to the system developer to select the one that is the most appropriate for the system. Several factors may influence the selection process. Among the most prominent factors, we foresee the existence of implementation for all or part of the dependable components embedded in the architectures. At this time, the selection of the most appropriate dependable architectures among the set of eligible ones is left upon the system developer.

We are currently examining solutions to help the developer in the selection process by coupling the architecture repository with an implementation repository. The benefit of our proposal for the construction of dependable systems lies in providing a repository of dependable architectures whose behaviors are precisely characterized using temporal first order logic. This characterization allows: (*i*) to infer an architectural description from a property specification, (*ii*) to retrieve an architecture providing the dependability property targeted for a given system, and (*iii*) to use an architecture selected from the repository to know how to extend a base non-dependable system with appropriate fault tolerance mechanisms. However, we cannot expect system developers to carry out the proofs appertained to the management of the repository of dependable architectures. Tools must be provided to assist this management. These tools include:

- A tool for the inference of a dependable architecture from the specification of a dependability property.
- A tool for updating the repository and retrieving architectures. This tool subdivides into a tool for classical database management, and a theorem prover for implementing the database functions that are defined over dependability properties.

We are currently implementing the first tool as well as the one relating to database management, their features are direct from the presentation we made in this paper. From the standpoint of providing a theorem prover, we are currently examining existing provers (e.g., (Manna *et al.* 1994)) so as to reuse an existing one for our framework.

4. CONCLUSIONS

This paper has presented a framework aimed at easing the construction of dependable systems. The framework relies on the formal specification of dependability properties, using temporal first-order logic. The proposed specification of dependability properties allows to infer the dependable software architecture corresponding to a property, which characterizes the structure of a dependable system with respect to the fault tolerance technique enforcing the given property. The structure of a dependable architecture further makes clear how to compose a dependable system from a base system. Formal specification of dependability properties enables us to provide a repository of dependable architectures, which is organized according to the refinement relation holding over dependability properties. Our proposal relates to a number of research efforts of the software engineering domain. In particular, it builds on results in the area of architecture description languages, and of software reuse.

From the standpoint of existing ADLs, there have been many proposals based on formal techniques. However, these proposals aim at complementary goals to ours. For instance, objectives for ADLs based on formal techniques include comparison of architectural styles using the Z notation (Abowd *et al.* 1995), reasoning about interaction patterns of architectural styles using a CSP-based calculus (Allen 1997), comparison of architecture designs and proving properties with regard to a specific architecture using the chemical abstract machine model (Inveradi and Wolf 1995), verification of reconfiguration correctness of architectures using graph grammars (LeMetayer 1996), definition of executable prototypes for architectures using partially ordered set of events (Luckham *et al.* 1995), and correct stepwise refinement of architectures using first-order logic (Moriconi *et al.* 1995). The last reference appears to be the most closely related to our proposal. However, in this reference, the architectural refinement relates to preserving topological constraints of the architectural elements. On the other hand, we are concerned with characterizing the semantics of an architecture from the standpoint of provided dependability properties. This characterization further serves to provide developers with a repository of dependable architectures that show how to make a base system dependable, using a fault tolerance technique enforcing the targeted dependability.

There is a significant amount of work in the area of software reuse (Krueger 1992). In this subsection, we concentrate on two research efforts on this topic: systematic component retrieval, and software reuse for customizing execution environment. To our knowledge, systematic component retrieval based on a specification of components using first-order logic has firstly been experimented in the Inscape environment (Perry 1989). This environment belongs to the family of development environments that can be seen as ancestors of the ones based on ADL, i.e., applications are described using a module interconnection language which is roughly an

ADL without the connector notion. The Inscape environment demonstrated that it was feasible to use the specification of components in terms of pre- and post-conditions to guide complex system design but also to retrieve component implementations in a systematic way. Successors of this proposal then enhanced the practicality of systematic software retrieval. A software retrieval tool that may be used in any development environment is presented in (Rollins and Wing 1991). This capacity is further enhanced in (Zaremski and Wing 1997), which provides a framework to support the definition of various refinement relations. Efficiency of software retrieval is addressed in (Mili *et al.* 1997), which proposes to organize the software database according to a refinement relation over software specifications. This work and its more recent version (Jilani et al. 1997) supply, moreover, a retrieval function that returns a software component approaching a specification if there is no available component matching the requested specification. The proposal presented in (Schumann and Fischer 1997) also addresses efficiency of the software retrieval process; it consists of using rejection filters based on signature matching and model checking technology to rule out non-matching components as early as possible. Our proposal builds on the above results and applies them to the domain of retrieving a software architecture with respect to a requested dependability property instead of a functional one.

Customizing execution platforms so as to adapt to application needs is now a growing concern in the software engineering domain. This has led to the definition of notations to ease the development of customized systems using existing software. Examples of environments offering such a facility can be found in (Batory and O'Malley 1992, Hiltunen and Schlichting 1995, Struman and Agha 1994). These proposals differ from ours in that we are addressing customization of execution platforms based on the refinement of requested dependability properties, while they provide a way to construct such platforms based on its adequate structuring. Thus, these environments could be conveniently exploited in our framework to take over the construction of the dependable system after the selection of the adequate dependable architecture.

REFERENCES

Abowd G. *et al.* (1995) Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319-364.

Allen R. (1997) A Formal Approach to Software Architecture. PhD Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, USA.

Allen R. and Garlan D. (1997) A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213-249.

Batory D. and O'Malley S. (1992) The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355-398.

Chrysanthis P. and Ramamritham K. (1994) Synthesis of Extended Transaction Models using Acta. *ACM Transactions on Database Systems*, 19(3):450-491.

Garlan D. *et al.* (1997) ACME: An Architecture Interchange Language. Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, USA.

Hiltunen M. A. and Schlichting R.D. (1995) Constructing a Configurable Group RPC Service. *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, pages 288-295.

Inverardi P. and Wolf A. L. (1995) Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373-386.

Jilani L. L. *et al.* (1997) Retrieving Software Components that Minimize Adaptation Effort. *Proceedings of the IEEE International Conference on Automated Software Engineering*, pages 255-262.

Krueger C. W. (1992) Software Reuse. *ACM Computing Surveys*, 24(2):131-183.

Lamport L. (1978) Time, Clocks, and the Orderings of Events in a Distributed System. *Communications of the ACM*, 21(7):558-565.

Laprie J. C. (1992) Dependability: Basic Concepts and Terminology. *Dependable Computing and Fault-Tolerant Systems*, Springer-Verlag.

LeMetayer D. (1996) Software Architecture Styles as Graph Grammars. *Proceedings of the ACM SIGSOFT'96 Symposium on Foundations of Software Engineering*, pages 15-23.

Luckham D. C. *et al.* (1995) Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336-355.

Manna Z. *et al.* (1994) STeP: The Stanford Temporal Prover. Technical Report No.94-1518, Computer Science Department, Stanford University, Stanford, CA, USA.

Mili R. *et al.* (1997) Storing and Retrieving Software Components: A Refinement Based System. *IEEE Transactions on Software Engineering*, 23(7):445-460.

Moriconi M. *et al.* (1995) Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356-372.

Perry D. E. (1989) The Inscape Environment. *Proceedings of the 11th International Conference on Software Engineering*, pages 2-12.

Perry D. E. and Wolf A. L. (1992) Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40-52.

Rollins E. J. and Wing J. M. (1991) Specifications as Search Keys for Software Libraries. *Proceedings of the 8th International Conference on Logic Programming*, pages 173-187.

Schumann J. and Fischer B. (1997) NORA/HAMMR: Making Deduction-based Software Component Retrieval Practical. *Proceedings of the 12th IEEE International Conference on Automated Software Engineering*, pages 246-254.

Shaw M. and Garlan D. (1996) Software Architecture: Perspectives on an Emerging Disciplines. Prentice Hall.

Stoller S. D. and Schneider F. B. (1996) Automated Analysis of Fault-Tolerance in Distributed Systems. Technical Report No.14853, Department of Computer Science, Cornell University, Ithaca, NY, USA.

Sturman D. C. and Agha G. A. (1994) A Protocol Description Language for Customizing Failure Semantics. *Proceedings of the Thirteenth IEEE Symposium on Reliable Distributed Systems*, pages 148-157.

Zaremski A. M. and Wing J. M. (1997) Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333-369.

ARCHITECTURAL MODELS AND DESCRIPTIONS

Specification and Refinement of Dynamic Software Architectures

Calos Canal, Ernesto Pimentel[1], José M. Troya
Depto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Spain
E-mail: {canal, ernesto, troya}@lcc.uma.es

Key words: Software architecture, architecture description languages, π-calculus, compatibility, inheritance of behaviour, prototyping

Abstract: Several notations and languages for software architectural specification have been recently proposed. However, some important aspects of composition, extension, and reuse deserve further research. These problems are particularly relevant in the context of open systems, where system structure can evolve dynamically, either by incorporating new components, or by replacing existing components with compatible ones. Our approach tries to address some of these open problems by combining the use of formal methods, particularly process algebras, with concepts coming from the object-oriented domain. In this paper we present LEDA, an Architecture Description Language for the specification, validation, prototyping and construction of dynamic software systems. Systems specified in LEDA can be checked for compatibility, ensuring that the behaviour of their components conforms to each other and that the systems can be safely composed. A notion of polymorphism of behaviour is used to extend and refine components while maintaining their compatibility, allowing the parameterisation of architectures, and encouraging reuse of architectural designs.

1. INTRODUCTION

The term software architecture (SA) has been recently adopted referring to the discipline of Software Engineering that deals with the description,

[1] This work was funded in part by the "Comisión Interministerial de Ciencia y Tecnología" (CICYT) under grant TIC98-0445-C03-03.

verification, and reuse of the structure of software systems (Shaw and Garlan, 1996). At the level of abstraction of SA, software is represented as a collection of computational and data elements, or components, interconnected in a certain way, and it is at this level where the structural properties of software systems are naturally addressed. SA pays special attention to the interaction among components, instead of the internal computations of these components.

The significance of explicit architectural specifications is widely accepted. First, they raise the level of abstraction, facilitating the description and comprehension of complex systems. Second, they increase reuse of both architectures and components (Shaw and Garlan, 1995). However, effective reuse of a certain architecture often requires that some of its components can be removed, replaced, and reconfigured without perturbing other parts of the application (Nierstrasz and Meijler, 1995). These aspects are particularly relevant when dealing with open distributed systems, whose architecture evolves dynamically, and consistency has to be guaranteed for every substantial change produced on the system. In the context of SA, consistency must be analysed in terms of the compatibility between components, since system performance depends on the correct interaction among them.

Although object-orientation can be applied to all levels of software design, in SA the more general term *component-oriented* is preferred, allowing to consider not only objects but architectures, interaction mechanisms and design patterns as first-class concepts of an architecture (Nierstrasz, 1995). However, most concepts coming from the object-oriented paradigm can be applied to SA. Particularly, in this work we address the application of inheritance, parameterisation, and polymorphism to the specification of software architectures.

A number of Architectural Description Languages (ADLs) have been already proposed. ADLs address the need for expressive notations in architectural design, trying to provide precise descriptions of the *glue* for combining components into larger systems. Despite the proposed notations are useful for the description of complex software systems, most of them are not formally based, which prevents the analysis and proof of the properties of the systems and architectures described (Abowd et al., 1993). In addition, several significant issues, such as specification of dynamic systems, architecture parameterisation and refinement, are not usually addressed. In Section 9 we compare our approach with other related works, particularly Wright and Darwin, while an exhaustive comparison on the characteristics of some outstanding ADLs can be found in (Medvidovic and Rosenblum, 1997).

In order to avoid some of these limitations, our interests focus on the application of formal methods to SA. Formal specifications have a precise meaning derived from the semantics of the notation used, and validation tools can be developed to prove properties of the systems specified. To this effect, process algebras are widely accepted for the specification of software systems, which can be checked for equivalence, deadlock freedom, and other interesting properties.

Dynamic architectures are those which describe how components are created, interconnected, and removed during system execution, and which allow run-time reconfiguration of their communication topology. Formal specification of such systems requires the use of an adequate formalism. In particular, we propose the use of the π–calculus (Milner et al., 1992), a simple but powerful process algebra which can express directly *mobility*, allowing the specification of dynamic systems in a very natural manner. However, the π–calculus is a low level notation, which makes difficult its direct application to the specification of large systems.

This was our original motivation for the development of LEDA, an ADL which embodies mechanisms of inheritance and dynamic reconfiguration. The language is structured in two levels: *components*, representing system parts or modules, and *roles*, which describe the observable behaviour of components. Roles are written in an extension of the π–calculus, thus allowing the specification of dynamic architectures. Each role describes the protocol that a component follows in its interaction with other components. In turn, components are described as composed of other components. The structure or architecture of a component is indicated by the relations among its subcomponents, which are expressed by a set of *attachments* or connections among the roles of these subcomponents.

LEDA differs from other ADLs in that it makes no distinction, at the language level, between components and connectors, i.e. connectors are specified as a special kind of components. This allows the language to be more simple and regular, and does not impose a particular compositional model in the description of software architectures.

Since the semantics of LEDA is written in terms of the π–calculus (Canal et al., 1998b), specifications can be both executed, allowing architecture prototyping, and analysed. In this sense, it is possible to determine whether a system is safely *composable*, i.e. whether its components present compatible behaviour and can be combined to form the system. This kind of analysis has been traditionally limited to interface conformance, but we are also interested in determining whether the *behaviour* of a component is compatible with that of its environment. On the other hand, component reuse would be encouraged if we could check whether a certain existing component can be used in a new system where a similar behaviour is required. Again, the intuitive notion of compatibility arises. We have formalised compatibility of behaviour in the context of π–calculus (Canal et al., 1998a), ensuring that compatible roles are able to interact successfully until they reach a well-defined final state. Architectures written in LEDA are tested for compatibility in each of the attachments among roles of their components. Compatibility does not require that the components involved have strictly complementary behaviour, since we usually want to connect components which match only partially.

Reuse of existing software components would be promoted if we had a way for adapting a component to an interface which is not compatible with its own interface. This is what LEDA adaptors are made for. Adaptors are small elements, similar to roles and also written in π–calculus, which are able to communicate successfully components whose behaviour is not compatible.

Our approach is completed with mechanisms of inheritance and parameterisation for roles and components which ensure that compatibility is preserved. A child component inherits its roles from its parents, while redefinition of behaviour is restricted by several conditions which ensure the maintenance of compatibility. Thus, we can replace safely a component in an architecture with any other component which inherits from the former. This gives place to a mechanism of architecture instantiation, by which a software architecture can be considered as a generic framework, which can be partially instanced and reused as many times as needed. Component frameworks derive from the idea of design patterns, and they represent the highest level of reusability in software development: not only source code of components, but also architectural design is reused in applications built on top of the framework (Pree, 1996). In this sense, LEDA specifications can be considered as generic architectural patterns or frameworks which can be extended and reused, adapting them to new requirements (Canal et al., 1997).

Although specification certainly plays an important role during system design and prototyping, the final goal of software design is to obtain real executable applications. LEDA specifications are also used for the creation, interconnection and deletion of components on an executable distributed platform. Combining the capabilities of prototyping and execution of LEDA, it is possible to simulate the execution of partially implemented systems. Hence, system development can be done gradually, providing a smooth transition from specification to implementation.

The structure of this paper is as follows. First, we describe briefly the π–calculus and the notation we use for specifying roles with it. Then, Sections 3 and 4 deal with the specification of components, roles and attachments in LEDA. Next, in Section 5 we discuss how our approach addresses architecture prototyping and validation, while Section 6 deals with component and role inheritance, and also addresses the topic of architecture refinement. Section 7 shows how non-compatible components can be interconnected using adaptors. All the notions introduced in these sections are illustrated by several examples. Finally, Section 8 discuss briefly how LEDA specifications can be used in order to derive executable applications from an architecture. We conclude comparing our approach with some related proposals.

2. THE π–CALCULUS

The π–calculus, developed by Milner as a successor of CCS, is specially suited for the description of dynamic systems, in which components are created and interconnected during system execution, because it permits direct expression of mobility. Mobility is achieved in π–calculus by the transmission of channel names as arguments or *objects* of messages. When a process receives a channel name, it can use this channel as a *subject* for future transmissions. This allows an easy and effective reconfiguration of the system. In fact, the calculus does not distinguish between channels and data, all of them are generically *names*. This homogeneous treatment of names is used to construct a very simple but powerful calculus. In contrast, π–calculus is a low level notation, and its use in industrial-size problems would be tedious and difficult.

LEDA embodies the π–calculus for specifying the roles which describe the behaviour of components. Roles are described in LEDA as processes, using a syntax which derives from the original notation of the π–calculus, adding some syntactic sugar to obtain more friendly specifications. Let *P,Q,...* range over processes, and *a,b,c,...* range over names. Sequences of names are abbreviated using tildes ($\tilde{a}$). Then, processes are built from names and processes as follows:

$$P ::= \mathbf{0} \mid \tau.P \mid x!(\tilde{o}).P \mid x?(\tilde{a}).P \mid (x)P \mid [x=z]P \mid P|Q \mid P+Q \mid A(\tilde{a})$$

Empty or inactive behaviour is represented by ***0***. Silent transitions, given by τ, model internal actions. Thus, a process *τ.P* will eventually evolve to *P* without interacting with its environment. An output-prefixed process $x!(\tilde{o}).P$ sends the names $\tilde{o}$ (objects) along name *x* (subject) and then continues like *P*. An input-prefixed process $x?(\tilde{a}).P$ waits for some names $\tilde{a}$ to be sent along *x* and then behaves like $P\{\tilde{o}/\tilde{a}\}$, where $\{\tilde{o}/\tilde{a}\}$ is the substitution of $\tilde{a}$ with $\tilde{o}$.

Restrictions are used to create private names. Thus, in *(x)P*, the name *x* is private to *P*. Private names can be exported to other processes simply by sending them as objects of output actions, as in *(z)x!(z)*. A match *[x=z]P* behaves like *P* if the names *x* and *z* are identical, and otherwise like ***0***.

The composition operator is defined in the expected way: $P \mid Q$ consists of P and Q acting in parallel. Summation is used for specifying alternatives: $P + Q$ may proceed to P or Q. The choice can be locally or globally taken. In a global choice, two processes agree synchronously in the commitment to complementary actions, as in

$$(\ldots + x!(\tilde{o}).P + \ldots) \mid (\ldots + x?(\tilde{a}).Q + \ldots) \rightarrow P \mid Q\{\tilde{o}/\tilde{a}\}$$

On the other hand, local choices are expressed combining the summation operator with silent actions. Hence, a process like $(\ldots + \tau.P + \tau.Q + \ldots)$ may proceed to P or to Q with independence of its context. We use local and global choices to state the responsibilities for action and reaction.

Finally, $A(\tilde{a})$ is an agent with names $\tilde{a}$. Each agent identifier A is defined by an unique equation: $A(\tilde{a}) = P$. The use of agents allows modular and recursive definition of processes.

Some examples of processes written in π–calculus can be found in the following sections, but for a detailed description of the calculus, including its transition system, we refer to (Milner et al., 1992).

3. COMPONENTS AND ROLES

LEDA is an ADL for the description and validation of structural and behavioural properties of software systems. The language is structured in two levels: *components* and *roles*. Components represent software pieces or modules, each one providing a certain functionality while roles describe the behaviour of components and are used for architecture validation, prototyping, and execution.

3.1 Components

LEDA distinguishes between component classes and instances, and provides mechanisms for the extension and parameterisation of components. The specification of a component class consists of three main sections: *(i)* **interface**, consisting of several role instances; *(ii)* structure or **composition**, consisting of several component instances; and *(iii)* **attachments**, which contains a list of connections which indicate how the component is built from its parts.

The interface of a component is described as a set of role instances, which specify the behaviour of the component from the point of view of each other component that interacts with it. Each role is a partial abstraction representing both the behaviour that the component offers to its environment, and the behaviour that it requires from those connected to it.

LEDA distinguishes between role classes and instances, and provides constructions for the extension and derivation of roles.

For instance, consider a file transmission between two components, named *Sender* and *Receiver* respectively. Component *Receiver* plays the role of *reader*, receiving the data which is sent by *Sender*, which acts as *writer* (*Figure 1*).

```
component Sender {                component Receiver {
    interface                         interface
        writer : Writer;                  reader : Reader;
}                                 }
```

Figure 1: Components Sender and Receiver

3.2 Specification of component's behaviour

Traditionally, interface description has been limited to the signature of the methods that a component imports and exports, or the messages that it can send or receive. However, our goal is to describe the observable behaviour of components, that is, how they react to external stimuli, and how input and output stimuli are related. This behaviour is described by the roles that form the interface of the component. Roles are specified as processes in the π–calculus.

Roles *Writer* and *Reader* in *Figure 2* specify the protocol of interaction between the components *Sender* and *Receiver*, i.e. they describe how these components behave in order to perform a successful data transmission. Data is transmitted matching two complementary actions *w!(data)* and *w?(data)*. As indicated by the use of local choices, the responsibility for action falls in the *Writer* part, which knows when the file has been completely transmitted, and sends an event *wq!()* (writer quits), while the *Reader* must be able to react to both *Writer* actions.

```
role Writer(w,wq) {                       role Reader(w,wq) {
 spec is                                   spec is
      τ.(data)w!(data).Writer(w,wq)             w?(data).Reader(w,wq)
    + τ.wq!().0;                              + wq?().0;
}                                         }
```

Figure 2: Roles Writer and Reader, from components Sender and Receiver

3.3 Composites

Components can be either simple or composite. A composite contains several subcomponents which are instances of other component classes. Any software system can be described as a composite. Thus, the syntax of LEDA does not distinguish between components and systems or architectures. As we have shown, simple components are described by the roles of their interfaces, but for composites, we must also describe their internal architecture. This architecture is the result of the interconnection or attachment of several subcomponents. The specification of composites in LEDA will be shown by means of a set of examples of increased complexity, describing a family of systems following a Client/Server architectural pattern.

Consider first a very simple Client/Server system in which the *Client* requests services from the *Server* (*Figure 3*). Both the *Client* and the *Server* are composites which contain an unbound array of service components. Role *request* describes the behaviour of the *Client*, while role *serve* describes that of the *Server*. When receiving a request, the *Server* creates a *service* component with the statement **new**. Then, the reference to the service is transmitted to the *Client* through the private link *reply*. Notice that the type of the component *service* is not indicated, but is declared of a generic type **any**, allowing future refinement of the Client/Server architecture, as will be shown in Section 6, for providing different kinds of services. The name *n* is used in the role *serve* for taking account of the number of requests received, which will be also used in a subsequent example.

```
component Client {
 interface
  request : Request(request) {
   spec is
    (reply)request!(reply).
       reply?(service).Request(request);
 }
 composition
  service[] : any;
 }
```

```
component Server {
 interface
  serve : Serve(request) {
   names
    n : Integer := 0;
   spec is
    request?(reply).
     (new service)reply!(service).
     n++.Serve(request);
   }
 composition
  service[] : any;
 }
```

Figure 3: Components Client and Server with their roles

4. ATTACHMENTS

The architecture of a composite is determined by the relations that its subcomponents maintain with each other. These relations are explicitly represented in LEDA by a set of attachments among the roles of these subcomponents. Attachments relate roles of several components, and they are specified in the composite which contains these components. Attachments are set when the corresponding components and role instances are created, possibly dynamically, and can be modified during system execution.

LEDA distinguishes among several kinds of attachments, which permit the specification of both static, reconfigurable, and dynamic software systems.

Static attachments are those which are never modified once they are set. For instance, recall the components *Client* and *Server* from *Figure 3*. We can specify our Client/Server architecture as a composite which contains both components and connects their roles using a static attachment (*Figure 4*). The symbol used for indicating the attachment is <>.

```
component ClientServer {
 interface none;
 composition
  client : Client;
  server : Server;
 attachments
  client.request(r) <> server.serve(r);
 }
```

Figure 4: A simple Client/Server system

On the other hand, *reconfigurable* attachments are used for architectures that present several configurations, i.e. those in which the interconnection patterns among components changes over time, and the roles connected depend on a certain condition. For instance, suppose that we have two *Server* components, and each request is assigned to one of them trying to balance their work load (*Figure 5*).

```
component ReconfigurableClientServer {
 interface none;
 composition
  client : Client;
  server[2] : Server;
 attachments
  client.request(r) <> if ( server[1].n <= server[2].n )
                       then server[1].serve(r)
                       else server[2].serve(r);
 }
```

Figure 5: A reconfigurable system, consisting of one Client and two Servers

Finally, *multiple* attachments describe communication patterns among arrays of components. Each pair of interconnected components may use private links in their communication, or these links may be shared by all the components involved. Thus, multiple attachments can be either shared or private. A shared attachment describes a 1:M communication channel, while private attachments establish multiple 1:1 communication channels.

For instance, consider a more realistic Client/Server system in which several *Clients* are connected to a pool of *Servers*. The composite *ServerPool* (*Figure 6*, left) contains an array of *Servers* whose roles are tied together using a multiple shared attachment (represented by the '*' in the left part of the attachment), and exported as a single role *serve* (role exportation is described below). Each request will be served by one of the *Servers* non-deterministically. On the other hand, the attachment between the *Clients* and the *ServerPool* is also multiple (*Figure 6*, right), and all clients share the link *r* through which they request services. Notice that mobility is used to establish private *reply* links for each request, though all the *Clients* are connected to the *ServerPool* using a single *request* link. Such an example can be hardly specified using formalisms like CSP (and consequently with CSP-based ADLs like Wright), which shows the richer expressiveness of the π–calculus when compared with other process algebras.

```
component ServerPool {                    component MultipleClientServer {
  interface                                       interface none;
    serve : Pool;                           composition
  composition                                 client[] : Client;
    server[] : Server;                        pool     : ServerPool;
  attachments                               attachments
    server[*].serve(r) >> serve(r);           client[*].request(r) <> pool.serve(r);
  }                                         }
```

Figure 6: A Client/Server system, with multiple clients and a pool of Servers

An additional form of attachment is that of role exportation. Usually, when dealing with a composite, some of the roles of its components are not used for the interconnection of these components, but to form the interface of the composite. Thus, we say that these roles are exported by the composite, which is indicated in LEDA using the operator >> instead of <>. We have already used this mechanism in *Figure 6*, left, where the roles of the *Servers* were exported to form the interface of the *ServerPool*.

5. ARCHITECTURE PROTOTYPING AND VALIDATION

The specifications written in LEDA can be used for prototyping. Attachments have a formal semantics (Canal et al., 1998b) which allows the derivation of π–calculus prototypes from architectural specifications. These prototypes can be executed using a π–calculus interpreter like the MWB (Victor, 1994). Thus, specifications can be tested at an early stage of the development process, checking their conformance with system requirements.

Apart from description and prototyping, LEDA specifications also serve for validation purposes. In particular, for determining whether a system is consistent, i.e. whether the behaviour of its components is compatible.

As we usually want to connect components that match only partially, the relations of bisimilarity customarily used in process algebras are not well suited for our purposes. Thus, we have defined a relation of role compatibility in the context of π–calculus. A formal definition of compatibility and its properties is out of the scope of this paper, but it can be found in (Canal et al., 1998a). A proof of compatibility for every system attachment using this relation ensures that the corresponding components will be able to interact safely until they reach a well-defined final state. Thus, if a software system is built according to the specifications of the architecture, no failure will arise from the interaction in any attachment between its components.

Obviously, local analysis of compatibility cannot ensure that the whole system is deadlock-free, since deadlock could arise from the global interaction of a set of components whose roles are compatible. However, compatibility serves for determining whether two components can be composed or plugged into each other, guaranteeing that the connector <> is safe. We consider that a system is consistent when each attachment in its architecture connects compatible roles, indicating behavioural conformance of the corresponding components. On the other hand, a failure detected when analysing an attachment stands for a mismatch in the behaviour of the corresponding components, usually leading to a system crash.

6. EXTENSION AND REFINEMENT

6.1 Extension of roles and components

In order to promote effective reuse of both components and architectures, a mechanism of redefinition and extension for roles and components is required. In the object-oriented paradigm, reuse is achieved by inheritance and polymorphism. Data polymorphism is defined as the capability of an identifier to point or refer to instances of different classes, while inheritance refers to a relation among object classes by which an heir class inherits the features (methods and attributes) of its parent classes. Heirs can extend their parents by adding new features, and they may also redefine some of the inherited features, usually under certain restrictions. Inheritance is a natural precondition for polymorphism, since it ensures that heirs will have at least the same features than their parents, and that they can replace them safely.

A relation of inheritance will be also of use for specifications of software components. However, in our context the interface of a component is defined not only by the signature of its features (i. e. the signature of its roles), but it also includes the behavioural patterns described in the roles. Thus, role redefinition and extension must be restricted in order to preserve the behaviour specified in the parent role. We have defined a relation of inheritance among roles in the context of π–calculus. This relation defines the restrictions for polymorphism of behaviour, allowing the replacement of a role by a derived version, while preserving compatibility. Role extension in LEDA can be formally validated. Again, we refer to (Canal et al., 1998a) for a formal definition of role inheritance and its properties.

Role extension can be used to *(i)* redefine, partially or completely, the parent role, giving a new specification for some of its agents; and *(ii)* extend a role, providing it with additional functionality. In both cases we must check, using the relation of inheritance, that the extended role is effectively an heir of the parent role.

For instance, consider the role *Serve* of *Figure 3*. Its behaviour can be extended allowing clients to query the number of requests solved by the server, which can be used for statistics.

```
role StatServe(request,statistics) extends Serve {
  adding
    statistics!(n).StatServe(request,statistics);
  }
```

Figure 7: An extension of role Serve

The notion of extension can be also applied to components. Derived components inherit their parent's specification, including roles, subcomponents and attachments. An heir component extends its parent by adding new roles, components, or attachments, or by redefining some of its parent's. In case of redefinition of a role or component, the redefined instance must be an heir of the original instance.

Component extension can be implicitly achieved by *architecture instantiation*, which indicates the replacement of a component instance in a composite with another one whose class extends that of the former. Architecture instantiation can be used for incremental specification, description of families of software products sharing a common architecture, and also for dynamic replacement of a component in a software system. The syntax of instantiation is as follows:

derivedComponent : ComponentClass[subcomponent : DerivedSubcomponentClass];

which means that *derivedComponent* is an instance of *ComponentClass* in which we have replaced its *subcomponent* (which let's suppose was declared of a certain *SubcomponentClass*) by an instance of *DerivedSubcomponent-Class*, where *DerivedSubcomponentClass* must be an heir of *Subcomponent-Class*.

When instancing an architecture, some of its attachments are modified, since some of its former components are replaced by derived versions. However, compatibility rechecking of the instanced architecture is not required, since role inheritance ensures the preservation of compatibility.

6.2 Architecture Refinement

Architectural descriptions can be used with different levels of abstraction during the development process. This property is commonly referred to as refinement. For example, we can begin with a high level specification of a system in which we describe only its top-level components, their interface, and how they are attached to construct the system. Then, refinement is applied to obtain a more detailed specification, by describing the internal structure or the behaviour of previously defined components, obtaining more complex specifications which come gradually closer to implementation. As we have seen, component extension is a useful mechanism for refinement, but other forms of refinement can be applied using LEDA.

In the Client/Server system in *Figure 6*, services were defined as generic components of type **any**. Thus, we have described an abstract Client/Server architecture which follows a simple protocol of requests and replies. We can obtain more specific architectures by describing the details of the service, i.e. describing the behaviour that both components follow during the service.

```
component ReceiverClient extends Client {
 interface
  request : RequestSenders(request) extends Request {
   spec is
    (reply)request!(reply).
     ( new receiver)reply?(service).RequestSenders(request);
   }
 composition
  receiver[] : Receiver;
  service[]  : Sender;
 attachments
  receiver[].reader(w,wq) <> service[].writer(w,wq);
}
```

Figure 8: Specialisation of a Client/Server, using Senders and Receivers

The *ReceiverClient* in *Figure 8* is a specialisation of the *Client* in *Figure 3*. Its role *request* is refined indicating that a component *receiver* is created each time the client requests a service. The *service* itself is refined, too, indicating that its type is now *Sender* instead of **any**, and a new attachment is included, connecting the roles of the *receiver* and the *service*. Components *Receiver* and *Sender* were specified in *Figure 1*, while its roles were described in *Figure 2*.

Hence, we have refined our Client/Server architecture, obtaining the description of a system in which the service provided is a file transmission. We can use the mechanism of architecture instantiation for obtaining an instance of the refined architecture:

refinedCS : MultipleClientServer[client : ReceiverClient, pool.server[].service[] : Sender];

Since role *RequestSenders* extends *Request*, compatibility with server's role *Serve* is ensured. On the contrary, the compatibility of the new attachment between the roles *Reader* and *Writer*, which was not present in the original architecture, must be checked.

7. ADAPTORS

Sometimes the behaviour of two components is not compatible, but these components can be adapted so they can collaborate with each other. This will be done using an adaptor, which acts as a glue allowing the construction of composites from components which are not strictly compatible. Adaptors are also used to modify the interface that a certain component exports to its environment.

Adaptors are specified in π–calculus, using the same syntax as for roles. However, roles describe the interface of a component, and they are declared in the interface section, while adaptors are mainly used as a glue to tie the components of a composite, and they are declared in the composition section.

In the preceding examples, servers are always prepared to receive requests, which is not a realistic assumption. The specification of a non-reliable server *NRServer* is shown in *Figure 9*, left. Observe how local choices, indicated by the combination of the sum operator and τ-transitions, specify that the *NRServer* may crash unexpectedly.

Obviously, the behaviour of our *NRServer* is not compatible with that of *Clients*, which suppose that servers are always willing to attend their requests. However, using a simple adaptor *restart* we can build a fault-tolerant server pool (*FTServerPool*, *Figure 9*, right). Each time an *NRServer* crashes it is restarted by the adaptor (in fact, it creates a new *NRServer*).

Thus, the adaptor modifies the observable behaviour of the pool of *NRServers*, and the combination of roles *NRServe* and the adaptor *Restart* provides an interface which can be proved as a refinement of role *serve* in *Figure 6*. Thus, *FTServerPool* extends *ServerPool*, and its behaviour is also compatible with role *Request*.

```
component NRServer {
 interface
   serve : NRServe(request,crash) {
     spec is
       τ.request?(reply).
         ( new service)reply!(service).
           NRServe(request,crash)
       + τ.crash!().0;
     }
 composition
   service[] : any;
 }
```

```
component FTServerPool extends ServerPool {
 composition
   server[] : NRServer;
   restart  : Restart(crash) {
     spec is
       crash?()( new server)Restart(crash);
     }
 attachments
   restart(e),server[*].serve(r,e) >> serve(r);
 }
```

Figure 9: A fault-tolerant pool of servers, built from non-reliable servers

Hence, we can instance the Client/Server architecture of *Figure 6* replacing its component *ServerPool* by an instance of *FTServerPool*:

ftcs : MultipleClientServer[pool : FTServerPool];

Compatibility with client's role *request* is ensured by inheritance, and there is no need to recheck the attachment between the server pool and the clients. Thus, we obtain a specialised version of the Client/Server system in which we use non-reliable servers, but maintaining the properties of the original architecture.

8. SYSTEM CONSTRUCTION AND EXECUTION

We have already discussed how LEDA specifications can be used for system validation and prototyping, but we can go one step further, and use them also for obtaining an executable system.

Using LEDA we can validate that each attachment in an architecture connects compatible roles. Our goal is now to translate this compatibility to the implementation level. First, each role is automatically translated into a state machine which encapsulates the behaviour of the corresponding component. These implementations of roles control the interaction of the corresponding components with the rest of the system. Thus, they are similar to IDL specifications, but augmented with the protocol that describes the behaviour of the components.

In turn, composites are responsible for the creation of components and for interconnecting their roles, following the communication patterns described in their attachments. Communication between roles is done using a process communication mechanism, (e.g. sockets).

Finally, components must be implemented using a programming language. Typically each component specification will be implemented as a class or group of classes using an object-oriented language. Each component is connected to its roles, through which it communicates with the rest of the system. When a component requires to invoke a method of another one, it invokes the corresponding method in its own role, which will contact the role of the other component in order to invoke the method.

Consider again the Client/Server system specified in *Figure 3*. Components *Client* and *Server* are implemented as classes, while their roles are translated into *RoleRequest* and *RoleServe* respectively (*Figure 10*, top).

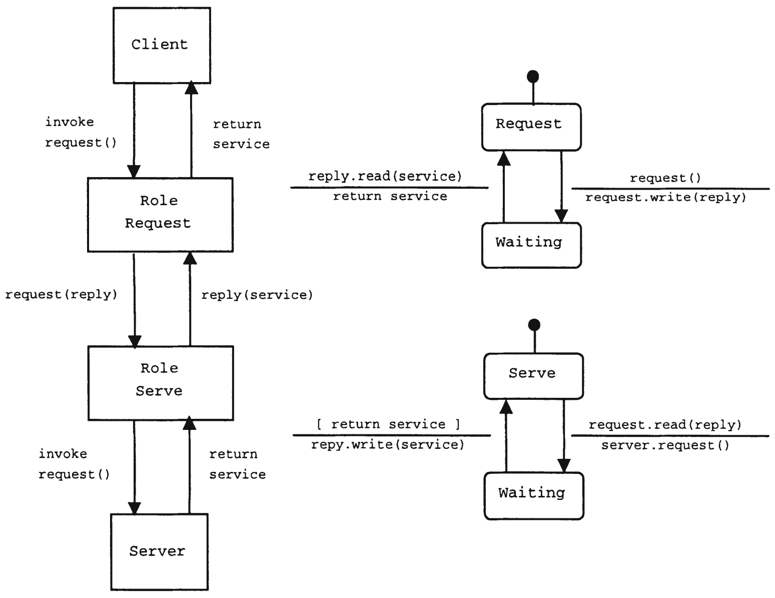


Figure 10: Implementation scheme of the Client/Server architecture

In order to obtain a service, the object *Client* invokes the method *request()* from its role *RoleRequest*. Then, *RoleRequest* sends the request to *RoleServe* through the appropriate channel. Next, *RoleServe* invokes the method *request()* from *Server*, and gets the service returned. The service is sent through a specific *reply* channel to *RoleRequest*, which in turns returns the service to *Client*. Thus, the implementations of *Client* and *Server* invoke or are invoked by their roles, but they don't know the location of the objects which finally receive the invocation, nor they are responsible for establishing or managing the communication channels indicated in the architecture.

This scheme for system implementation has several advantages. First, connections among components are encapsulated in the roles, which establish and modify them according to the interaction patterns specified in the architecture. Second, components are implemented as object classes that invoke or receive invocations of methods, but which are independent of the interaction mechanisms used in the architecture. Third, a component may have several implementations which can be interchanged without affecting the behaviour of the system.

9. DISCUSSION

In this paper we have presented LEDA, an ADL for the description of dynamic software architectures. In these systems, components interact following flexible patterns that can be modified during system execution. The basic unit in LEDA is that of components, which are represented by their interface, divided into a set of roles. These roles describe, using the π–calculus, the behaviour of the corresponding components. Software architectures are specified in LEDA as sets of components related by attachments between their roles. The semantics of components and attachments is given using the π–calculus, a well-known process algebra, which allows us to use this formalism for architecture prototyping and validation of properties like behavioural compatibility. LEDA roles and components can be extended, adapting them to new requirements, but maintaining the compatibility of the original roles. Analysis of compatibility and inheritance can be both automated, which leads to the development of tools for the analysis of the specifications. Formal validation of compatibility and inheritance encourage both software quality and reuse, determining whether some existing software components can be used to build a larger system.

In the last years several proposals related to the specification of software architectures have been presented. Although most of them are not formally founded, which limits their possibility of analysis, several works have already proposed the use of different formalisms for architecture specification.

A first formalisation of the notion of compatibility is described in (Allen and Garlan, 1997), where CSP is used for determining compatibility in the ADL Wright. However, formalisms like CSP or CCS do not seem appropriate for the description of evolving or dynamic structures. At most, CSP can be used in systems with a finite number of configurations, as it is shown in (Allen et al., 1998), but not in highly dynamic systems, where the π–calculus is best suited. Furthermore, Wright does not address aspects of component and role extension or refinement, nor of architecture simulation or execution.

Our approach differs from that of Allen and Garlan in other significant aspect: LEDA does not distinguish between components and connectors, nor between ports and roles. This distinction would complicate unnecessarily the language, specially the formalisation of compatibility and inheritance in π–calculus. Besides, we consider that the distinction between components and connectors does not scale properly, since composition would lead to mixed composites with free ports and roles which could not be considered either as components nor as connectors. For these reasons, connectors are described in LEDA as specific classes of components, their behaviour being described by roles.

The π–calculus has been used for describing the semantics of several computer languages. In fact, the operational semantics of the ADL Darwin (Magee and Kramer, 1996) is described using π–calculus, endowing this language with a mechanism of dynamic binding. However, type checking is restricted in Darwin to name matching, and the behaviour of components is not described, neither this language incorporates characteristics of extension or inheritance. On the contrary, our approach uses the π–calculus not only for semantics, but it integrates the calculus in the language. LEDA components and attachments are higher-level constructs that simplify the description of complex software systems, while LEDA roles take advantage of the expressiveness of the π–calculus for describing the behaviour of components. This allows us to state more precisely which are the relations between the components of a certain software architecture, and also to perform analysis of compatibility and inheritance.

The notions of component subtyping and inheritance are present in several other ADLs, and recent work of (Medvidovic et al., 1998), addresses description and verification of behavioural conformance using the Z notation. On the contrary, our approach describes component's behaviour using state machines, and addresses what they call *protocol conformance*.

We are currently working in the development of a Java run-time platform for LEDA, capable to use the information about component behaviour and architecture configuration present in the specifications to create, interconnect and remove the implementations of the components described using the language, thus obtaining executable applications.

Our future work will be the application of LEDA to the specification of different industrial software systems, in order to determine the need for new forms of interaction in the language. Another task will be the development of supporting tools, such as graphic editors or validation tools. All these tools should hide the difficulties inherent to the formal foundations of the language, making easier the specification of software systems in LEDA to those not acquainted with formal methods.

REFERENCES

Abowd, G., Allen, R., and Garlan, D. (1993). Using style to understand descriptions of software architecture. In *Proc. ACM FSE'93*.

Allen, R., Doucence, R., and Garlan, D. (1998). Specifying and analyzing dynamic software architectures. In *Proc. ETAPS'98*, Lisbon.

Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM Trans. on Software Engineering and Methodology*.

Canal, C., Pimentel, E., and Troya, J. (1997). On the composition and extension of software systems. In *Proc. of FSE'97 FoCBS Workshop*, pp. 50–59, Zurich.

Canal, C., Pimentel, E., and Troya, J. (1998a). Compatibility, inheritance and extension of π–calculus agents. Technical Report LCC-ITI-98-13, Computer Science Dept., Universidad de Málaga. http://www.lcc.uma.es/~canal/LCC-ITI-98-13.

Canal, C., Pimentel, E., and Troya, J. (1998b). π–calculus semantics of an architecture description language. Technical Report LCC-ITI-98-17, Computer Science Dept., Universidad de Málaga. http://www.lcc.uma.es/~canal/LCC-ITI-98-17.

Magee, J. and Kramer, J. (1996). Dynamic structure in software architectures. In *Proc. ACM FSE'96*, pp. 3–14, San Francisco.

Medvidovic, N. and Rosenblum, D. (1997). Domains of concern in software architectures and architecture description languages. In Proc. USENIX Conf. on Domain-Specific Languages, Santa Barbara (USA).

Medvidovic, N., Rosenblum, D. and Taylor, R. (1998). A Type Theory for Software Architectures. Technical Report UCI-ICS-98-14. Dept. Information and Computer Science, University of California, Irvine.

Milner, R., Parrow, J. and Walker, D. (1992). A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77.

Nierstrasz, O. (1995). Requirements for a composition language. In *Proc. of ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution*, no. 924 in LNCS, pp. 147–161. Springer Verlag.

Nierstrasz, O. and Meijler, T. (1995). Research directions in software composition. *ACM Computing Surveys*, 27(2):262–264.

Pree, W. (1996). *Framework Patterns*. SIGS Publications.

Shaw, M. and Garlan, D. (1995). Formulations and formalisms in software architecture. In van Leeuwen, J., editor, *Computer Science Today*, no. 1000 in LNCS, pp. 307–323. Springer Verlag.

Shaw, M. and Garlan, D. (1996). Software Architecture. Perspectives of an Emerging Discipline. Prentice Hall.

Victor, B. (1994). A verification tool for the polyadic π–calculus. Master's thesis, Department of Computer Systems, Uppsala University (Sweden).

Modeling Software Architectures and Styles with Graph Grammars and Constraint Solving

Dan Hirsch, Paola Inverardi, and Ugo Montanari
Departamento de Computaión, Universidad de Buenos Aires, Ciudad Universitaria, Pab.I, (1428), Buenos Aires, Argentina, dhirsch@dc.uba.ar
Dip. Di Mat. Pura ed Applicata, Università dell'Aquila, Via Vetoio, Localita' Coppito, L'Aquila, Italia, inverard@univaq.it
Dipartimento di Informatica, Università di Pisa, Corso Italia 40, (56125), Pisa, Italia, ugo@di.unipi.it

Key words: Architectural descriptions, graph rewriting, styles, dynamic architectures, reconfiguration

Abstract: A software architecture style is a class of architectures exhibiting a common pattern. The description of a style must include the structure model of the components and their interactions (i.e., structural topology), the laws governing the dynamic changes in the architecture, and the communication pattern. A simple and natural way to describe a system is by using graphs, and as an extension of this, by using grammars to describe styles. The construction and dynamic evolution of the style will be represented as context-free productions and graph rewriting. To model the evolution of the system we use techniques of constraint solving already applied in the representation of distributed systems. From this approach we obtain an intuitive way to model systems, and a unique language to describe the style (but still a clear separation of coordination and configuration). With these we have a direct way of following the evolution of the system and proving its properties.

1. INTRODUCTION

A software architecture style is a class of architectures exhibiting a common pattern (Shaw, M. and Garlan, D., 1996). The description of a style must include the structure model of the components and their interactions (i.e., structural topology), the laws governing the dynamic changes in the architecture, and the communication pattern. In the following we refer to all these aspects as a *complete style* description. A simple and natural way to describe a system architecture is by using graphs, and as an extension of this, by using grammars to describe styles. So a grammar will generate all possible instances of that style. This approach has first been proposed in (Le M'etayer, D., 1998).

In our work we represent a system as a graph where edges (or hyperedges) (Drewes, F. et al., 1996) are components and nodes are ports of communication. The construction and dynamic evolution of the style will be represented as *context-free* productions and graph rewriting. The productions that represent the style will be grouped in three sets. The first one contains the productions that correspond to the construction of the initial static configuration of the system. The second set contains the rules that model dynamic changes in the configuration of the system (create and remove components) and the third set contains the rules that model the communication pattern.

To model the evolution of the system we need to choose a way of selecting which components will evolve and communicate. For this we propose a technique already applied in (Montanari, U. and Rossi, F., 1997) and (Montanari, U. and Rossi, F., 1996) to represent distributed systems with graph rewriting and constraint solving. A graph represents a distributed system, where edges represent processes and nodes represent shared data. In order to evolve, one process may need to synchronize with adjacent processes on some conditions on the shared data. If they agree on these conditions, then all of them can evolve. This is modeled by a two phased approach where, context-free process productions are specified (a set for each process) with synchronization conditions for each of the possible moves. After that, context-sensitive subsystem rewriting rules are obtained by combining some context-free productions (this is called *the rule-matching problem*) (Corradini, A. et al., 1985).

Applying one of these context-sensitive rules, allows for the evolution of a subpart of the system consisting of several processes (each with one of its context-free productions) that agree on the conditions imposed on the shared data. Applying the rule means making all such processes (and not a proper subset of them) evolve, each with one of its context-free productions.

The solution to the rule-matching problem is implemented considering it as a finite domain constraint problem (Mackworth, A., 1988). In this paper we will not describe these techniques; the interested reader may refer to the references. In the case of software architectures we use constraint rules to coordinate the dynamic evolution of the system. This is done by using constraints on ports to represent communication between components and (if necessary) to control changes in the configuration of the system. One difference from (Montanari, U. and Rossi, F., 1997), is that in our approach, we will rely on two basic types of communication paradigms: point-to-point and broadcast communication. These will be represented with two types of nodes. With point-to-point communication the rule-matching problem is easier; it has to choose only two rules (for each sender, one receiver). In the case of broadcast the solution is the same as in (Montanari, U. and Rossi, F., 1997). This allows to represent both types of communication at the same time.

The use of hyperedge rewriting grammars and constraints to represent styles and model evolution gives us an intuitive way to model systems and a unique language to capture the style, but still with a clear separation between coordination and configuration. Besides we have a direct way of following the evolution of the system and proving properties and the inheritance of the distributed solutions for the rule-matching problem. Moreover, context-free hyperedge rewriting is natural for modeling the behavior of components independently of each other, and its generality can be used (if one wants to) to incorporate descriptions of more complex connector elements in the specification of a system (you just represent connectors as edges and their evolution as productions).

A related work that uses graph grammars is (Le M'etayer, D., 1998). There, a dual approach is taken and architectural styles are represented as context-free graph grammars where nodes represent components and edges their communication links. But, in this case the grammar only specifies the static configuration of the system (referred to as the style). The dynamic evolution (create and remove components) is defined independently by a *coordinator*, and the rules of the coordinator are checked to preserve the constraints imposed by the grammar that defined the style. Also a CSP-like language for the individual entities is given to fit with the coordinator semantics.

The main difference between the two approaches is that in our work we give a uniform description of the *complete style* with grammars (but still maintaining an independent description of components behavior). Also, we don't have a global coordinator of evolution; instead, each component defines its own evolution (Magee, J. and Kramer, J., 1996a).

In (Le M'etayer, D., 1998), communication links are represented as edges, and components as nodes. We chose a dual approach, because we want the evolution of the style (including the communication pattern) to be modeled with the rewriting steps of the graphs. So, in this way hyperedges (and their associated nodes) are used only to represent components and the ports that they will share and use to communicate among them. A graph with this representation gives a simple view of the structure of an instance of architecture at a given state, separated from the application of the rewriting rules that shows the evolution between states. In this way, a clearer representation of the system is obtained while a separation of configuration and evolution is achieved, which is a desirable property of software architecture description languages (Medvidovic, N., 1997).

Another important point is that the evolution and communication pattern can be followed directly by the rewriting sequences on the graphs, analogously to what happens in the CHAM description of software architectures (Inverardi, P. and Wolf, A., 1995). This also allows the verification of properties of the architecture, such as deadlock (Degano, P. and Montanari, U., 1987; Compare, D. et al.,).

In section 2 basic notions of graph rewriting and constraint rules are introduced, then in section 3 we apply these notions to software architectures using some examples, and finally in section 4 we draw the conclusions and describe our future work.

2. BACKGROUND

In this section we introduce the basic notions of hypergraphs, hypergraph rewriting, and constraint productions.

2.1 Graphs and Graph Rewriting

DEFINITION [HYPERGRAPHS]

We define an edge-labeled hypergraph, or simply a graph as a tuple $G = \langle N, E, c, ext, lab_{LN}, lab_{LE} \rangle$, where:

1. N is a set of nodes.
2. E is a set of edges.
3. $c: E \rightarrow N^*$ is the connection function (each edge can be connected to a list of nodes).
4. $ext \in N^*$ is a set of external nodes.
5. $lab_{LE}: E \rightarrow LE$ is the labeling function of edges.
6. $lab_{LN}: N \rightarrow LN$ is the labeling function of nodes.

A *graph production* rewrites a graph into another graph, deleting some elements (nodes and edges), generating new ones, and preserving others. In this paper we will just consider context-free productions, which rewrite a graph containing a single hyperedge, into an arbitrary graph, while preserving the (*external*) nodes connected by the rewritten hyperedge. Therefore, in a context-free production, no nodes are deleted.

DEFINITION [GRAPH PRODUCTIONS]

Given a set of external nodes *EN*, a graph production *p* is a pair < L , R >, where:

1. *L* is a graph containing only an hyperedge.
2. *R* is a graph.
3. The external nodes of *L* and *R* are exactly those in *EN*.

Context-free graph productions will be written as $L \rightarrow R$, where *L* is the (graph containing only the) hyperedge to be rewritten and *R* is the graph to be generated. A production $p = (L \rightarrow R)$ can be applied to a graph *G* yielding *H* $(G \Rightarrow_p H)$ if there is an *occurrence* of *L* in *G*. The result of applying *p* to *G* is a graph *H* which is obtained from *G* by removing the occurrence of *L* and adding *R*.

DEFINITION [GRAPH REWRITING SYSTEM]

A graph rewriting system is a pair $GRS = < G_0 , P >$, where:

1. G_0 is a graph.
2. *P* is a set of graph productions.

A derivation for *GRS* is a finite sequence of direct derivation steps of the form $G_0 \Rightarrow_{p1} G_1 \Rightarrow_{p2} \ldots \Rightarrow_{pn} G_n = H$, where $p_1, \ldots, p_n$ are in *P*.

To model coordinated rewriting, it is necessary to add some labels to the nodes in the left member of productions. Assuming an alphabet of requirements A, we need a partial function $f: nodes(L) \rightarrow A$ that associates conditions (or actions) to some of the nodes. In this way, each rewrite of an edge must match conditions with its adjacent edges and they have to move as well. For example, consider two edges that share one node, such that no other edge is attached to that node, and let us take one production for each of these edges. Each of these productions has a condition on that node (*a* and *b*). If $a \neq b$, then the edges cannot rewrite together (using that rule). If $a = b$, then they can move, via the context-sensitive rule obtained from merging the two context-free rules (*rule-matching problem*).

3. GRAPH REWRITING FOR SOFTWARE ARCHITECTURE STYLES

Now we will apply the notions introduced in the previous section to the description of software architectures. Software architectures are represented as hyperedge graphs where edges are components and nodes are communication ports. Two edges sharing a node means that there is a communication link between the two components. As we mentioned in the introduction, we have two types of nodes: point-to-point and broadcast communication.

A software architecture style is described by a hyperedge context-free grammar. The productions of a grammar are grouped in three sets.

The first set represents the construction of all possible initial configurations of the class of architectures modeled by the style.

The second set represents the rules for the dynamic evolution of the configuration, this means create and remove components.

The third set contains the rules that model the communication pattern of the architecture. This set contains productions to model the communication evolution for each type of component. These rules are constrained productions that during rewriting will coordinate for the evolution of the system. Also, some of the rules in the second set can be (if necessary) constrained. This can be used to model coordinated changes in the configuration. We will show this in the second example.

Edge labels have two parts. One is the component name and the other is the status of the component that represents its different states during evolution. Edges are drawn as boxes, broadcast ports as full circles, and point-to-point ports as empty circles. Nodes are labeled with port names (port names are local to rules, and external nodes have to be matched when a production is applied). Constraints decorate nodes in bold letters, and appear on the right-hand part of a production. For point-to-point we have a CCS like notation for the constraints, where a node labeled as $\bar{a}$ means that the component is the sender of a message a and a node labeled a is its receiver. For broadcast, all nodes that have to coordinate are labeled with the constraint representing the message.

Now we present three simple examples to show how a style is modeled.

3.1 Client-Server

The first example is a client-server case study based on the one used in (Le M'etayer, D., 1998). We have clients, servers and a manager. An instance of the style can have an initial configuration with any number of clients, any number of servers and one manager. Clients and servers communicate through the manager. Clients and manager are connected via the *CR* (client request) and *CA* (client answer) ports. Servers and manager are connected via the *SR* (server request) and *SA* (server answer) ports. In this example all nodes are point-to-point ports.

As we said at the beginning of this section we grouped productions in three sets. The first set represents the construction of all possible initial configurations of the class of architectures modeled by the style.

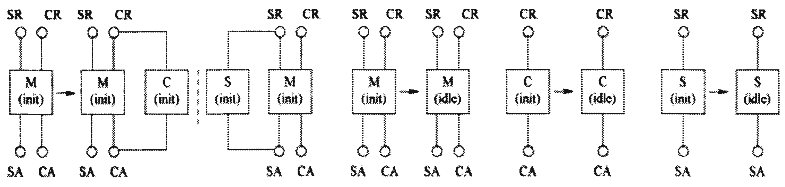


Figure 1. Client-server: static productions

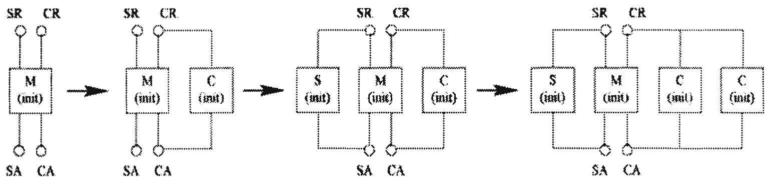


Figure 2. Client-server: an instance of the architecture style generated by the static productions

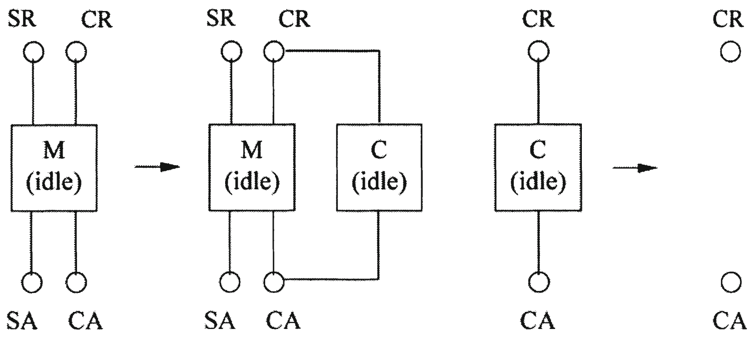


Figure 3. Client-server: dynamic productions

For the client-server example these are the productions in figure 1. This figure shows that all instances start with the manager and then clients and servers are attached to it. This is done by the application to the manager of the first and second rules in figure 1 (the dashed line is a shortcut to describe two productions for the manager). Note that the status of all components at this level is *(init)*, indicating that they are in a construction (or initialization) phase.

Figure 2 shows an instance with two clients and one server generated by these productions. After the desired initial configuration is obtained, then (init) → (idle) rules are applied (last three in figure 1). These rules mean that the construction phase is over and that the system is ready to start to work. Now, you can apply the last two sets of rules for the evolution of the architecture.

Figure 3 shows the dynamic rules. In this example we have two simple rules. The first one states that the manager accepts the incorporation of a new client in the system, and the second one is for clients that want to leave the system.

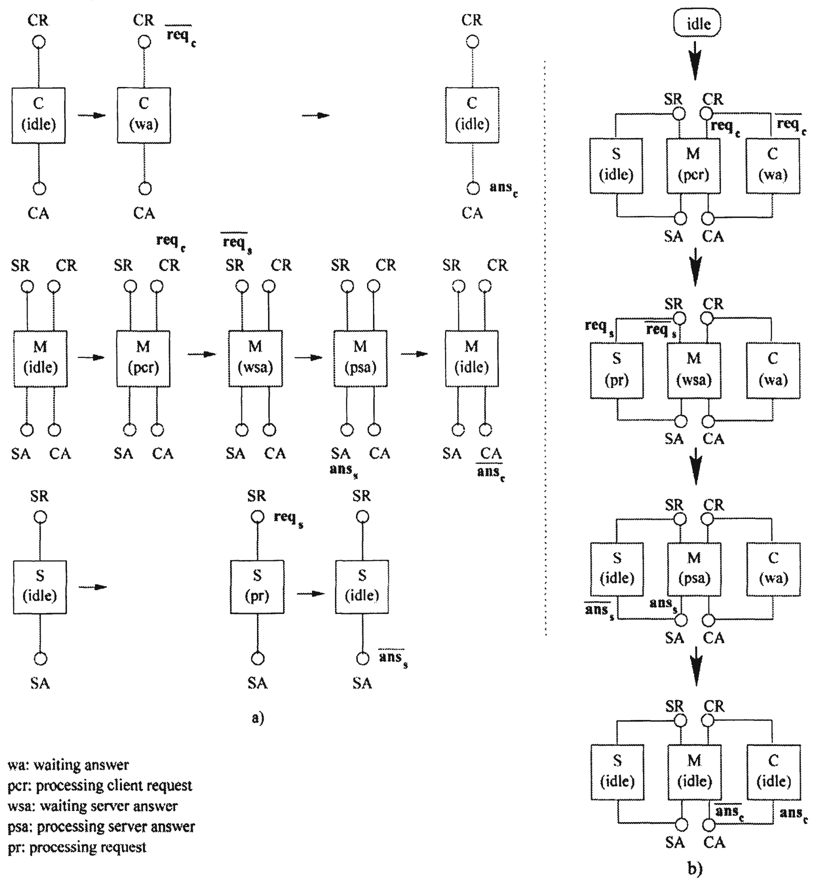


Figure 4. Client-server: communication pattern productions

Figure 4*a* shows the rules corresponding to the communication pattern. Note that all component specifications are independent from each other and that the only relation between them is by the communication coordination.

This is important for a better understanding and analysis of the system behavior. In this example all ports are point-to-point so, the manager will have to choose among the clients that want to make a request (obviously this is handled by the constraint resolution algorithms). In a broadcast communication all rules that want to rewrite and share nodes have to agree on the conditions imposed by the constraints.

In figure 4*b* you can see how the constrained rules work with a client that sends a request, the manager, and a server that returns the answer. These components can be part of a bigger graph but we assume that they were already chosen by the constraint solving algorithm at each rewriting step. The three components start from an *idle* state. Then the manager and the client rewrite respectively to the *pcr* and *wa* states after having coordinated on the client request. The second rewriting is between the manager and the server (to *wsa* and *pr* states, respectively) when the manager forwards the request the server. The last two steps are from the server to the manager (to *idle* and *psa* states, respectively) delivering the answer, and from the manager to the client returning the answer to its request. At the end of the sequence they return to an *idle* state (the server already after returning the answer), where new communications can be performed or any of the dynamic productions can be applied.

Note that the dynamic productions in figure 3 can be applied only when components are in an *idle* status (they cannot be in the middle of a communication).

It is worthwhile mentioning that we choose the level of abstraction for the description of the communication pattern. For example, figure 5*a* is an alternative set of rules for the communication pattern, where there are two rewrites instead of four: one that sends the request from the client to the server (via the manager), and the other that returns the answer to the client (figure 5*b*).

With this grammar we obtained a complete characterization of the style in a unique language and a clear identification of the steps that every architecture instance gives during its evolution. Also note that by analyzing the derivation tree it is possible to have all the computations of the system allowing the verification of properties of the architecture, such as deadlock (Degano, P. and Montanari, U., 1987).

3.2 Remote Medical Care System

This example is a simplification of a case study presented in (Balsamo, S. et al., 1998) for performance evaluation of a software architecture. We present here only a partial specification of the style, to show how constraints

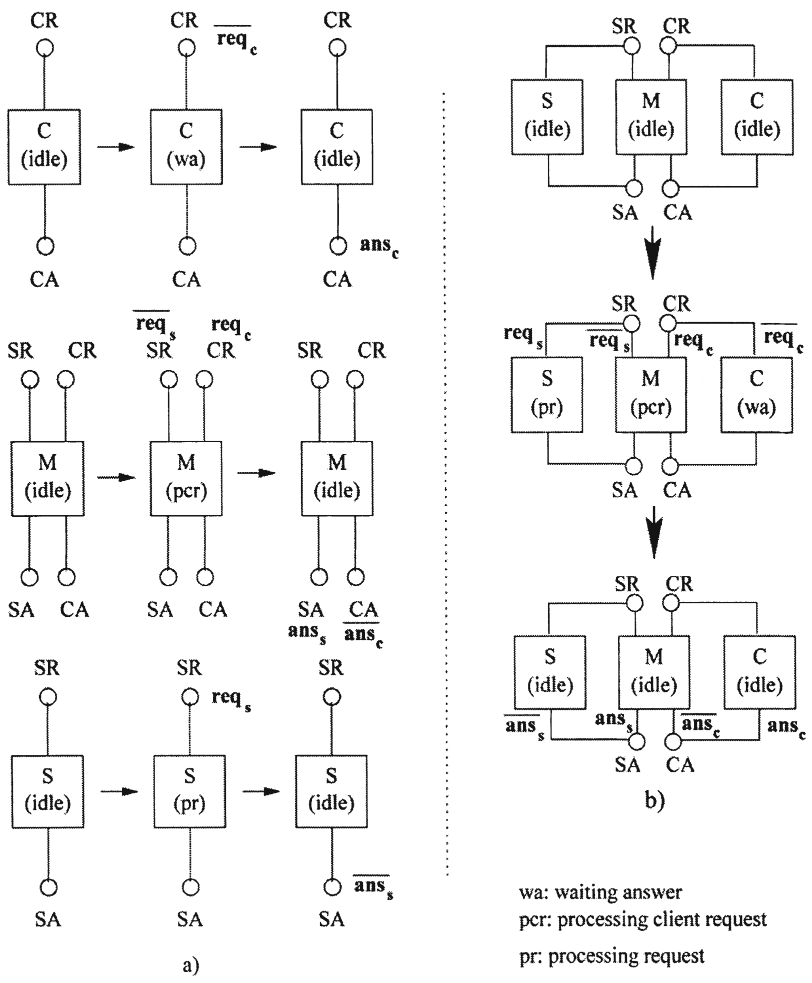


Figure 5. Client-server: communication pattern productions—an alternative

can be used to control an ordered evolution in the configuration of the system. This system is part of a project carried out by the University of L'Aquila at Parco Scientifico e Tecnologico d'Abruzzo, a regional consortium of public and private research institutions and manufacturing industries.

The Teleservices and Remote Medical Care System (TRMCS) provides and guarantees assistance services to users with specific needs, like disabled or elderly people. It is composed of a set of *Users*, which are connected to a

Router which interacts with a *Server*. An external component, the *Timer* allows the modeling of time.

The four types of units operate as follows:

- **User** sends either alarm (i.e., help requests) or check signals (i.e., control messages about the subsystem user state or the user's health state, respectively).

- **Router** accepts signals (control or alarm) from the users. It forwards the alarm requests to the Server and checks the behavior of the subsystem user though the control messages.

- **Server** dispatches the help requests.

- **Timer** sends a clock signal for each time unit.

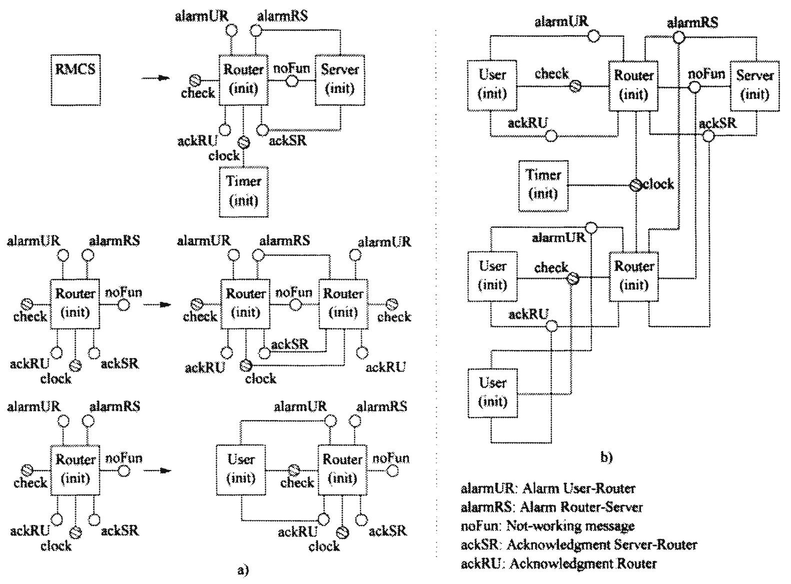

RMCS
Router (init)
Server (init)
Timer (init)
User (init)
alarmUR
alarmRS
noFun
check
ackRU
ackSR
clock
a)
b)
alarmUR: Alarm User-Router
alarmRS: Alarm Router-Server
noFun: Not-working message
ackSR: Acknowledgment Server-Router
ackRU: Acknowledgment Router

Figure 6. TRMCS: static productions

There is only one server in the system. A variable number of routers are connected to the server and a variable number of users are connected to each router. The timer controls all routers. Figure 6*a* shows the static productions and figure 6*b* shows an instance of the system with two routers, and one user attached to the first one and two others to the second router.

In the Client-Server example presented above, clients can leave the system independently of the other components. The only restriction, as it is modeled in the productions, is that they cannot leave the system if they are in the middle of a communication. In the TRMCS system, users have a similar behavior to the clients, but for routers the situation is different. In the case of a router, it is allowed to leave the system, but it cannot disappear without checking if there are users still connected to it. One possible action for the router if there are users connected to it, is to wait until all of them leave and then, when there are no users connected, it can leave too. These actions are described with the productions in figure 7.

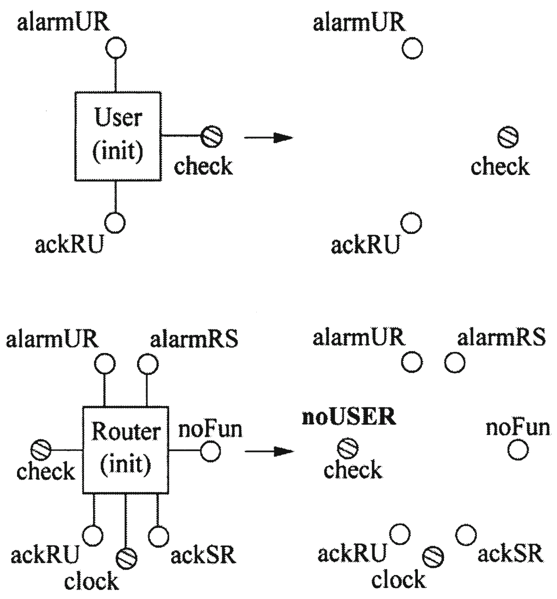


Figure 7. TRMCS: dynamic productions

These productions are part of the set of dynamic productions for the TRMCS. The first rule is for the user and it allows it to leave the system independently (i.e., it is not constrained). The second rule is for the router and it is constrained. The condition **noUSER** is imposed on the *check* port. A router and its users are connected to this port. This is a broadcast type of port, so a condition in it means that for this rule to be applied it must coordinate with all other edges connected to that port (i. e. the users). So, if all neighbors agree on the condition, then everybody can rewrite. But in this case, the only one with this condition is the router and it cannot leave the system while users are attached to it.

When all users connected to the router leave the system then the production with condition **noUSER** is satisfied and then it can be applied to the router. The rule can be applied because there are no neighbors, so the router is the only one that has to agree on the constraint. Note that, after the router leaves the system, the three isolated nodes that remain can be eliminated with a special action called *end* that functions as a type of garbage collection. This is an example of coordinated evolution, where constraints are used to control and coordinate the dynamics of a system.

3.3 Connectors: Parallel Point-to-point

Software architectures may require complex interactions among components. Usually, connectors may be defined as architectural building blocks to help model and specify these interactions. The modeling of connectors explicitly and independently, helps to achieve a higher level of reusability allowing to use already specified connectors in different styles and to create new connector types as the composition of basic ones.

So, in this direction we propose to use the generality of the model we are presenting to obtain independent connector descriptions. Using the same language to specify connectors based on more basic ones, allows to incorporate them to the primitive set of communication types and reuse them successively in different style descriptions.

In all the examples presented we use two basic types of communication: broadcast and point-to-point communication. In the next example we use the broadcast port as a basic type and specify with constrained productions the parallel point-to-point communication. The specification of a parallel point-to-point port allows for a set of adjacent components to perform parallel communications between pairs. This means that in a given port (the one we are specifying), for each sender a receiver (if there are available) is selected to accept the communication and if there are more than one pair willing to communicate, simultaneous interactions are allowed.

Figure 8 shows the specification for a connector (edge *C*) from two to four components. In this figure all ports are broadcast ports. For two components broadcast and point-to-point are the same (figure 8*a*). For three components, figure 8*b* shows the three possible alternatives (with three components there are no parallel communications) and figure 8*c* shows the nine possible interactions that can take place between four components. In a similar way this specification can be generalized for *n* components. In this case, point-to-point communication is associative and commutative so once we have the connector specification we can abstract from it and use the new connector as a new type of port. Also, we can mention that repeatedly composing the connector specification for three components, and the corresponding one for four components (only considering the rules for a single pair communication), in a sequential pattern, we obtain the simple point-to-point communication. In this way, an independent specification of a new connector is obtained and it can be reused in the description of other software architecture styles.

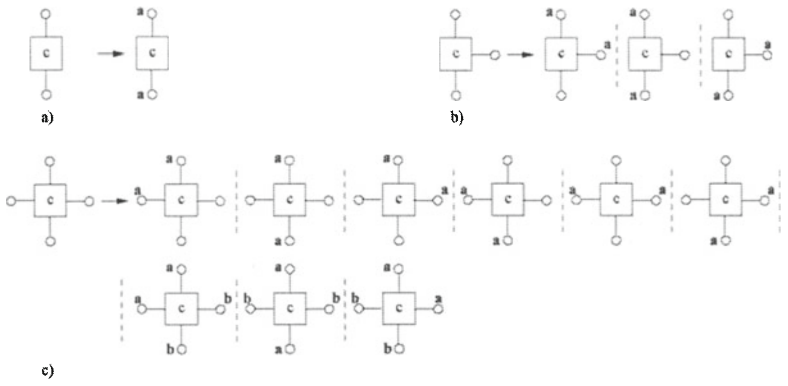


Figure 8. Parallel point-to-point connector

4. CONCLUSIONS AND FUTURE WORK

In this work we have presented a specification method for software architecture styles using hyperedge context-free graph grammars. Based on the rewriting system specified by the grammars we describe the style as a set of productions that model the initial structural topology of the architecture, the laws governing the dynamic changes, and its communication pattern.

Among the benefits of this approach are: a simple description of systems with a unique language is obtained, the use of constraints to model coordination of components allows a clear description of component interactions and controlled dynamics, and the inheritance of the distributed solutions for the rule-matching problem. As we said, we propose to use a technique already applied in (Montanari, U. and Rossi, F., 1997) and (Montanari, U. and Rossi, F., 1996) to represent distributed systems with graph rewriting and constraint solving. This is modeled by a two-phased approach where, context-free process productions are specified (a set for each process) with synchronization requests for each of the possible moves.

After that, context-sensitive subsystem rewriting rules are obtained by combining some context-free productions.

The solution of the rule-matching problem is implemented considering it as a finite domain constraint problem (Mackworth, A., 1988), where variables are associated with processes and constraints with ports. The domain of a variable is then the set of all context-free productions for the corresponding process, and each constraint is satisfied by the tuples of context-free productions (one for each adjacent process) whose synchronization requirements agree on the considered port. In this kind of constraint problem, a solution is thus a choice of a context-free production for each process, such that all synchronization requirements are satisfied. Usually, finite domain constraint problems are solved by a backtracking search over a tree of the possible alternatives for each variable. To deal with this type of problems many efficient techniques have been proposed, such as constraint propagation or local consistency algorithms (Mackworth, A., 1988), (Dechter, R. and Pearl, J., 1988). As in (Montanari, U. and Rossi, F., 1997), this kind of graph rewriting can be raised to a general framework, called *the tile model* (Gadducci, F. and Montanari, U., 1996), that permits a clear separation between sequential rewriting and synchronization.

Also, context-free rules are a natural way for modeling the behavior of components independently of each other allowing a distributed implementation, and as we saw in the client-server example, constrained rules allows different levels of detail for the description of transactions (Bruni, R. and Montanari, U., 1997). This is a convenient property to model architectures in which components are required to configure themselves (Magee, J. and Kramer, J., 1996a).

In this paper we model ports just as connections between components but as was shown in the examples the generality of the method can be used to incorporate descriptions of more complex connector elements in the specification of a system. If it is necessary complex connectors can be incorporated as a new type of edge.

Another thing to note is that in the examples presented we did not include termination rules. Constraints and productions can be used to model local and coordinated termination and this will be important for the verification of properties on the derivation tree.

We agree that the use of context-free rules limits the type of architecture styles that can be described, but we consider this as a first step on our work. With this type of rules, two separate edges already created cannot be bound later, so for example, an architecture instance that has a pipeline style cannot be converted, after its creation, into a ring. This is a great restriction that can easily be modeled in languages like • -calculus. But work like (Montanari,

U. and Pistore, M., 1995), shows that this type of calculus can be represented with graph rewriting (not context-free).

Finally, the productions that we use are all rewriting rules (one thing is replaced by another), but an interesting extension is to incorporate refinement rules where the history of the system is remembered. It is worth mentioning that in the original paper (Degano, P. and Montanari, U., 1987) the partial ordering is generated with the past history of the derivation. This can be useful in the description of a bigger class of software architectures, specially those in which the organization of components and connectors may change during system execution (Magee, J. and Kramer, J., 1996b).

In spite of the fact that context-free productions limit the classes of systems that can be described, it is clear that the description language proposed has very good properties for modeling reconfiguration and self organising architectures. It is our intention to continue the research in this direction for a deeper analysis of the subject.

ACKNOWLEDGMENTS

The third author was partially supported by CNR Integrated Project Sistemi Eterogenei Connessi mediante Reti di Comunicazione, Esprit Working Group COOR-DINA and Italian Ministry of Research Tecniche Formali per Sistemi Software. The first author was partially supported by ARTE Project, PIC 11-00000-01856, ANPCyT and FOMECProject 376, Contract 164.

REFERENCES

Balsamo, S., Inverardi, P., Mangano, C. and Russo, F. (1998). Performance evaluation of a software architecture: A case study, *Proceedings of the Ninth International Workshop on Software Specification and Design*.

Bruni, R. and Montanari, U. (1997). Zero-safe nets, or transaction synchronization made simple, *EXPRESS'97, Electronic Notes in Theoretical Computer Science* **7**.

Compare, D., Inverardi, P. and Wolf, A. (n.d.). Uncovering architectural mismatch in dynamic behavior. To appear.

Corradini, A., Degano, P. and Montanari, U. (1985). Specifying highly concurrent data structure manipulation, in Bucci, G. and Valle, G. (eds), *COMPUTING 85:A Broad Perspective of Concurrent Developments*, Elsevier Science.

Dechter, R. and Pearl, J. (1988). Network-based heuristics for constraint satisfaction problems, in Kanal and Kumar (eds), *Search in Artificial Intelligence*, Springer Verlag.

Degano, P. and Montanari, U. (1987). A model for distributed systems based on graph rewriting, *Journal of the Association for Computing Machinery* **34**(2).

Drewes, F., Kreowski, H.-J. and Habel, A. (1996). Foundations, in G. Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation*, Vol. I, World Scientific, chapter 2.

Gadducci, F. and Montanari, U. (1996). The tile model, *Technical Report TR-96-27*, Department of Computer Science, University of Pisa.

Inverardi, P. and Wolf, A. (1995). Formal specification and analysis of software architectures using the chemical abstract machine model, *IEEE Transactions on Software Engineering* **21**(4): 373-386. Special Issue on Software Architectures.

Le M'etayer, D. (1998). Describing software architecture styles using graph grammars, *IEEE Transactions on Software Engineering* . to appear.

Mackworth, A. (1988). *Encyclopedia of IA*, Springer Verlag, chapter Constraint Satisfaction.

Magee, J. and Kramer, J. (1996a). Dynamic structure in software architectures, *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Software Engineering Notes.

Magee, J. and Kramer, J. (1996b). Self organising software architectures, *Proceedings of the Second International Software Architecture Workshop*.

Medvidovic, N. (1997). A classification and comparison framework for software architecture description languages, *Technical Report ICS-TR-97- 02*, University of California, Irvine, Department of Information and Computer Science.

Montanari, U. and Pistore, M. (1995). Concurrent semantics for the • -calculus, *Electronic Notes in Theoretical Computer Science* **1**.

Montanari, U. and Rossi, F. (1996). Graph rewriting and constraint solving for modelling distributed systems with synchronization, *Lecture Notes in Computer Science* **1061**.

Montanari, U. and Rossi, F. (1997). Graph rewriting, constraint solving and tiles for coordinating distributed systems. To appear in Applied Category Theory.

Shaw, M. and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall.

Describing Software Architecture with UML

C. Hofmeister, R. L. Nord, D. Soni
Siemens Corporate Research, Princeton, New Jersey, USA
{chofmeister, rnord, dsoni}@scr.siemens.com

Key words: Software architecture, UML, architecture descriptions, multiple views

Abstract: This paper describes our experience using UML, the Unified Modeling Language, to describe the software architecture of a system. We found that it works well for communicating the static structure of the architecture: the elements of the architecture, their relations, and the variability of a structure. These static properties are much more readily described with it than the dynamic properties. We could easily describe a particular sequence of activities, but not a general sequence. In addition, the ability to show peer-to-peer communication is missing from UML.

1. INTRODUCTION

UML, the Unified Modeling Language, is a standard that has wide acceptance and will likely become even more widely used. Although its original purpose was for detailed design, its ability to describe elements and the relations between them makes it potentially applicable much more broadly. This paper describes our experience using UML to describe the software architecture of a system.

For these architecture descriptions, we wanted a consistent, clear notation that was readily accessible to architects, developers, and managers. It was not our goal to define a formal architecture description language. The notation could be incomplete, but had to nevertheless capture the most important aspects of the architecture. In this paper we start by giving an overview of the kinds of information we want to capture in a software architecture description. Then we give an example of a software architecture description for part of particular system: the image processing portion of a real-time image acquisition system. The final section discusses the strengths and weaknesses of UML for describing architecture.

We separate software architecture into four views: conceptual, module, execution, and code. This separation is based on our study of the software architectures of large systems, and on our experience designing and reviewing architectures (Soni, 1995). The different views address different engineering concerns, and separation of such concerns helps the architect make sound decisions about design trade-offs.

The notion of this kind of separation is not unique: most of the work in software architecture to date either recognizes different architecture views or focuses on one particular view in order to explore its distinct characteristics and distinguish it from the others (Bass, 1998). The 4+1 approach separates architecture into multiple views (Kruchten, 1995). The Garlen and Shaw work focuses on the conceptual view (Shaw, 1996). Over the years there has been a great deal of work on the module view (Prieto-Diaz, 1986). There is other work that focuses on the execution view, and in particular explores the dynamic aspects of a system (Kramer, 1990; Purtilo, 1994). The code view has been explored in the context of configuration management and system building.

The conceptual view describes the architecture in terms of domain elements. Here the architect designs the functional features of the system. For example, one common goal is to organize the architecture so that functional features can be added, removed, or modified. This is important for evolution, for supporting a product line, and for reuse across generations of a product.

The module view describes the decomposition of the software and its organization into layers. An important consideration here is limiting the impact of a change in external software or hardware. Another consideration is the focusing of software engineers' expertise, in order to increase implementation efficiency.

The execution view is the run-time view of the system: it is the mapping of modules to run-time images, defining the communication among them, and assigning them to physical resources. Resource usage and performance are key concerns in the execution view. Decisions such as whether to use a link library or a shared library, or whether to use threads or processes are made here, although these decisions may feed back to the module view and require changes there.

The code view captures how modules and interfaces in the module view are mapped to source files, and run-time images in the execution view are mapped to executable files. The partitioning of these files and how they are organized into directories affect the buildability of a system, and become increasingly important when supporting multiple versions or product lines.

Each of the four views has particular elements that need to be described. The elements must be named, and their interface, attributes, behavior, and relations to each other must be described. Some of the views also have a configuration, which constrains the elements by defining what roles they can play in a particular system. In the configuration, the architect may want to describe additional attributes or behavior associated with the elements, or to describe the behavior of the configuration as a whole.

In the next four sections, we show how we used UML to describe each of these four views, starting with the conceptual view and ending with the code view. To make the explanation clearer, we use an example from an image acquisition system.

The image acquisition system acquires a set of digitized images. The user controls the acquisition by selecting an acquisition procedure from a set of predefined procedures, then starting the procedure and perhaps adjusting it during acquisition. The raw data for the images is captured by a hardware device, a "camera", and is then sent to an image pipeline where it is converted to images. The image pipeline does this conversion, first composing the raw data into discrete images, and then running one or more standard imaging transformations to improve the viewability of the images. The image pipeline is the portion of the system that we will use as an example.

2. CONCEPTUAL ARCHITECTURE VIEW

The basic elements in the conceptual view are components with ports through which all interactions occur, and connectors with roles to define how they can be bound to ports. The components and connectors are bound together to form a configuration. In order to bind together a port and role in a configuration, the port and role protocols must be compatible. Components can be decomposed into other components and connectors. These elements, their associated behavior, and the relations of the conceptual view are summarized in Table 1.

Table 1. Elements of a conceptual architecture view

Elements	*Behavior*	*Relations*
component	component functionality	component decomposition
port	port protocol	port-role binding (for configuration)
connector	connector protocol	
role	role protocol	

Figure 1 is a UML diagram that describes much of the conceptual view for the image pipeline. It is represented by the ImagePipeline component, which has ports acqControl for controlling the acquisition, packetIn for the incoming raw data, and framedOutput for the resulting images.

The ImagePipeline is decomposed into a set of components and connectors that are bound together to form a configuration. The components, ports, and connectors are a stereotype of Class[1], but we use the convention of special symbols for ports and connectors (and leave off the stereotype for components) in order to make the diagrams easier to read. Roles are shown as labels on the port-connector associations. We also use the convention that when an association's multiplicity is not specified, it is assumed to be one.

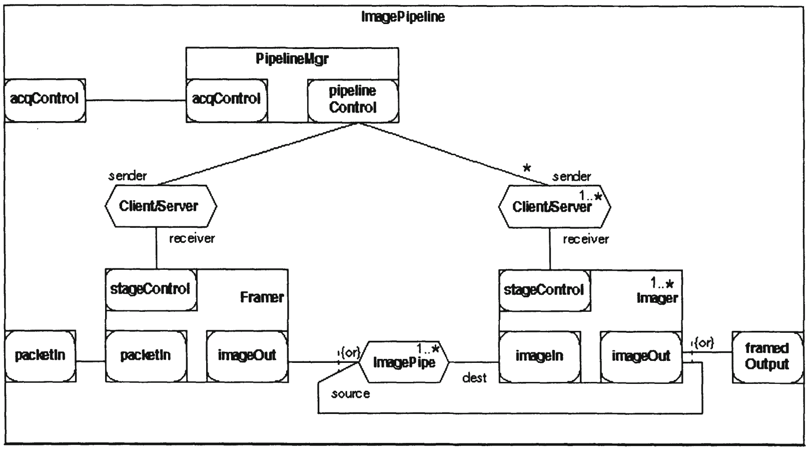


Figure 1. Conceptual configuration

The multiplicities on the components, connectors, and bindings show the set of allowable configurations. Each acquisition procedure has a distinct set of processing steps, represented by the Imager component. So the diagram shows the general structure of an image pipeline, which all acquisition procedures adhere to.

The first stage of the pipeline is the Framer, followed by one or more subsequent stages, represented by the Imager. Each of the stages is connected to

"A stereotype is, in effect, a new class of modeling element that is introduced at modeling time. It represents a subclass of an existing modeling element with the same form (attributes and relationships) but with a different intent... To permit limited graphical extension of the UML notation as well, a graphic icon or a graphic marker (such as texture or color) can be associated with a stereotype." (UML, 1997) the pipelineControl port via a Client/Server connector. The Imager component has a multiplicity of "1..*", meaning that an acquisition procedure has one or more of these later stages.

The Imager is bound to "1..*" Client/Server connectors, but the association is one-to-one, so each Imager instance is bound to exactly one Client/Server instance. Each Client/Server instance is bound to the pipelineControl port of exactly one PipelineMgr, but pipelineControl is bound to all Client/Server instances in the pipeline. Similarly the "1..*" ImagePipe connectors have a one-to-one association with the Imagers. Because the bindings also have multiplicities, we can conclude that there are the same number of Client/Server, Imager, and ImagePipe elements bound together in a legal configuration.

We use the "{or}" annotation at the source side of the ImagePipe to show that an ImagePipe is either bound to the output of the first stage or a later stage. But while the output of the first stage (the Framer) is always bound to the ImagePipe, the later stages could be bound to framedOutput. When a later stage is bound to framedOutput, it is necessarily the last stage in the pipeline.

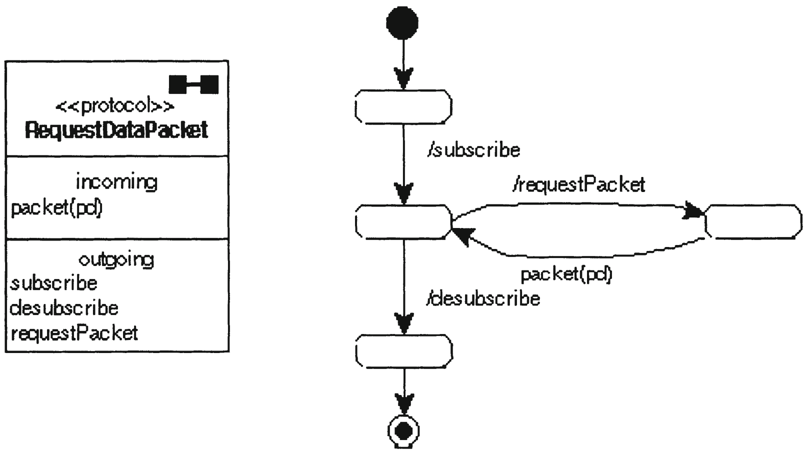


Figure 2. Protocol for packetIn Port

Figure 2 shows the protocol RequestDataPacket, which the packetIn ports on the ImagePipeline and Framer follow. We have adopted the ROOM notation here, showing the incoming and outgoing messages, then either a sequence diagram or state diagram to show the legal sequences of these messages (Selic, 1994; Selic, 1998).

The resource budgets are attributes of the components and connectors. They can be described in the attribute box of the appropriate class in a UML diagram, in a table, or in text.

For the conceptual view, we represent components, ports, and connectors as stereotyped classes. Decomposition is shown with nesting (association), and bindings are shown by association. We use:

- UML Class Diagrams for showing the static configuration.
- ROOM protocol declarations and UML Sequence Diagrams or State Diagrams for showing the protocols that ports adhere to.
- UML Sequence Diagrams for showing a particular sequence of interactions among a group of components.

3. MODULE ARCHITECTURE VIEW

In the module architecture view, subsystems are decomposed into modules, and modules are assigned to layers in accordance with their use-dependencies (Table 2). There is no configuration for the module view because it defines the modules and their inherent relations to each other, but not how they will be combined into a particular product.

Table 2. Elements of the module architecture view

Elements	*Behavior*	*Relations*
module	interface protocol	module implements
subsystem		conceptual component
layer		subsystem decomposition
		module use-dependency

Table 3 shows how the image pipeline's conceptual elements are mapped to module elements. Notice that ports, connectors, and components are sometimes combined into one module. This information could also be shown in a UML class diagram, with the mapping between conceptual and module elements shown as an explicit association.

Table 3. Mapping between conceptual and module architecture views

Conceptual element	*Subsystem or Module*
ImagePipeline	SPipeline
acqControl, pipelineControl	MPipelineAPI
PipelineMgr, ImagePipe, Client/Server	MPipelineControl, MImageBuffer
stageControl, imageIn, imageOut	MImageMgrAPI
Framer	MFramer
Imager	MImager

The SPipeline subsystem is decomposed into the six modules shown in Figure 3. This decomposition is dictated by the modules' correspondence to the conceptual elements, and their decomposition. Again we use nesting to show the decomposition, and we use stereotypes for each different type of element.

We do not use the UML "component" notation for a module, because in the module view the modules are abstract, not the physical modules of source code.

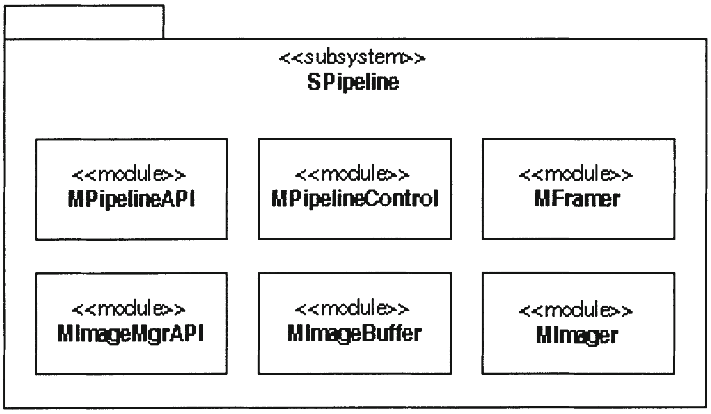


Figure 3. Decomposition of SPipeline

The use-dependencies among the pipeline modules are also derived from the conceptual elements' associations. These are shown in Figure 4. The MClient and MDataMgrAPI are not part of the SPipeline subsystem, but we included them in order to show all use-dependencies of the SPipeline subsystem. We use the UML "lollipop" notation to show the interface(s) of each module, and to make it clear that the modules are dependent on the interface of another module, not the module itself.

Figure 4 also shows some of the layers of the system. These are based on the use-dependencies among modules and subsystems, so we often show use-dependencies between and within layers in the same diagram, as we did here.

For the interface definition, we use a simple list of the interface methods. This information could be put inside the class definition in a UML diagram. We generally prefer to list it separately, using the class diagrams to focus on the relations among modules rather than a complete description of the modules. In the module view, we represent modules with a stereotyped class, and subsystems and layers with stereotyped packages. Decomposition is shown by nesting (association), and the use-dependency is a UML dependency.

We use:

– tables for describing the mapping between the conceptual and module views.

- UML Package Diagrams for showing subsystem decomposition dependencies.
- UML Class Diagrams for showing use-dependencies between modules.
- UML Package Diagrams for showing use-dependencies among layers and the assignment of modules to layers.

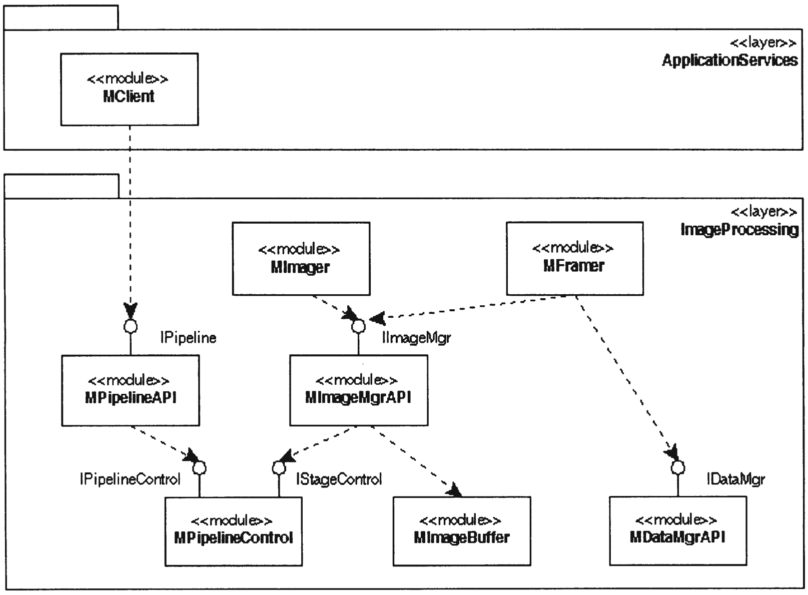


Figure 4. Use-dependencies of SPipeline

4. EXECUTION ARCHITECTURE VIEW

The execution architecture view describes how modules will be combined into a particular product by showing how they are assigned to run-time images. Here the run-time images and communication paths are bound together to form a configuration. Table 4 lists the elements, behavior, and relations of the execution view.

Table 4. Elements of the execution architecture view

Elements	*Behavior*	*Relations*
run-time image communication path	communication protocol	run-time image contains module binding (for configuration)

The execution configuration of the Image pipeline in Figure 5 indicates that there is always just one EClient process, but multiple pipelines can exist at one time. A pipeline has one process each for EPipelineMgr, EImageBuffer, and EFramer, and one process each for additional pipeline stages.

We again use a stereotype of the UML Class for run-time images. They are stereotyped with the name of the platform element, in this case <<process>> or <<shared data>>. We originally used the UML "active object" notation for a process, but now prefer to use a stereotyped class. One reason is that we often want to use classes rather than objects in a configuration diagram. A second reason is that active objects have a thread of control, whereas passive objects run only when invoked (UML, 1997). This distinction was not what we wanted to describe; we wanted to characterize the run-time image by its platform element (e.g. process, thread, dynamic link library, etc.) rather than convey control flow information about the elements.

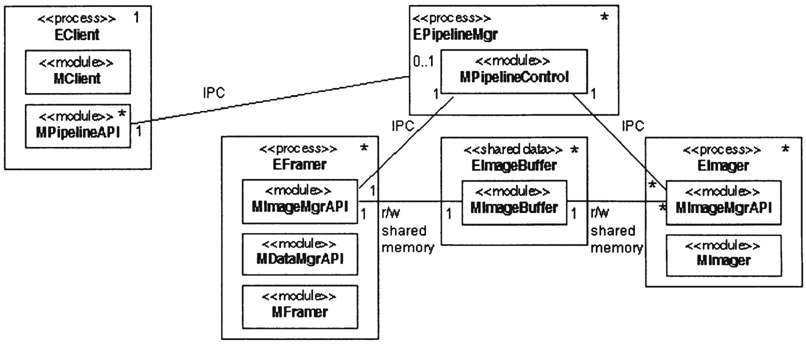


Figure 5. Execution configuration of the image pipeline

This diagram uses nesting to show the modules associated with each run-time image. The modules have a multiplicity that is assumed to be one if none is explicitly shown. In the configuration in Figure 5, there are multiple modules MImageMgrAPI, but at most one per process, and only in the EFramer and EImager processes. There are also multiple modules MPipelineAPI in the configuration, but all of these reside in process EClient.

The run-time images also have multiplicity, as do communication paths, which are labeled to show the communication mechanisms. This has the same implications as for the conceptual configuration, namely that with multiplicities on the run-time images, communication paths, and modules we can show all allowable configurations in a single diagram.

UML class diagrams cannot show dynamic behavior, so we use different diagrams to show the dynamic aspects of configurations. Figure 5 shows the configuration of the pipeline during an imaging procedure. The processes that implement the pipeline are created dynamically when the imaging procedure is requested, and are destroyed after the procedure has completed. A UML sequence diagram shows how the pipeline is created at the start of a procedure (Figure 6).

For the execution view, we represent the run-time images as stereotyped classes, and the communication paths as associations. Module containment is shown by nesting (association). We use:

- UML Class Diagrams for showing the static configuration.
- UML Sequence Diagrams for showing the dynamic behavior of a configuration, or the transition between configurations.
- UML State Diagrams or Sequence Diagrams for showing the protocol of a communication path.

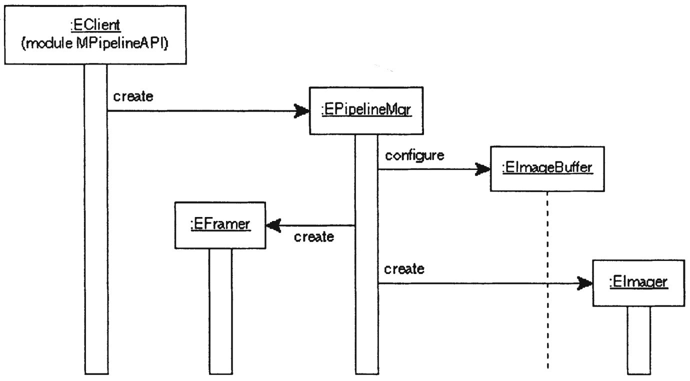


Figure 6. Image pipeline creation

5. CODE ARCHITECTURE VIEW

The code architecture view contains files and directories, and like the module view, does not have a configuration. The relations defined in the code view apply across all products, not just to a particular product. The code view elements and their relations are listed in Table 5. Modules and interfaces from the module view are partitioned into source files in a particular programming language.

Table 6 shows this mapping for the MPipelineControl module and its interfaces: the public interfaces are each mapped to a file, and we have created an additional file for the private interface to the module.

Table 5. Elements of code architecture view

Elements	*Relations*
source	source implements module
intermediate	source includes source
executable	intermediate compiled from run-time image
directory	executable implements run-time image
	executable linked from intermediate

Table 6. Source files for module MPipelineControl

Module or Interface	*Source File*
MPipelineControl	CPipelineControl.CPP, CPipelineControlPvt.H
IPipelineControl	CPipelineControl.H
IStageControl	CStageControl.H

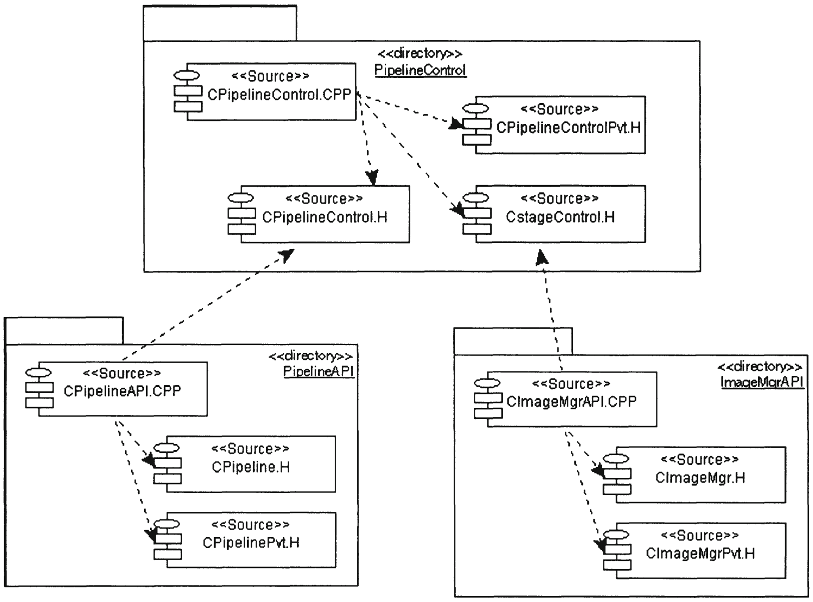


Figure 7. Include dependencies among source files

The source files are organized into directories, as shown in Figure 7. We use the UML "component" notation to represent the files, and the package notation for directories. Both files and directories have stereotypes to clarify their meaning. In UML, the component symbol is used for "source code components, binary code components, and executable components" (UML, 1997). We believe the intention of this symbol is closest to our notion of a file (whether source, intermediate, or executable).

Figure 7 also shows the include dependencies for the PipelineControl source files. We use the UML dependency notation for these relationships, with the stereotype <<include>> if the diagram contains more than one type of dependency. Source files can also have a "generate" dependency, for example when a preprocessor uses one source file to generate another.

The run-time images from the execution view also have a relationship to elements in the code view, in this case to executable files. Table 7 shows how two of the run-time images in the image pipeline are mapped to executable files. Here the mapping is one-to-one, but if the run-time image contained dynamic link libraries, each of these libraries would be in a separate executable file.

Table 7. Mapping between run-time image and executable file

Run-time Image	*Executable File*
EPipelineMgr	EPipelineMgr.exe
EFramer	EFramer.exe

The executable files are also organized into directories (Figure 8). The relationship between executable files and source files is through intermediate files. An executable file has link dependencies to the object files it links in, and an object file has compile dependencies to the source files from which it is compiled. These dependencies are also shown in Figure 8.

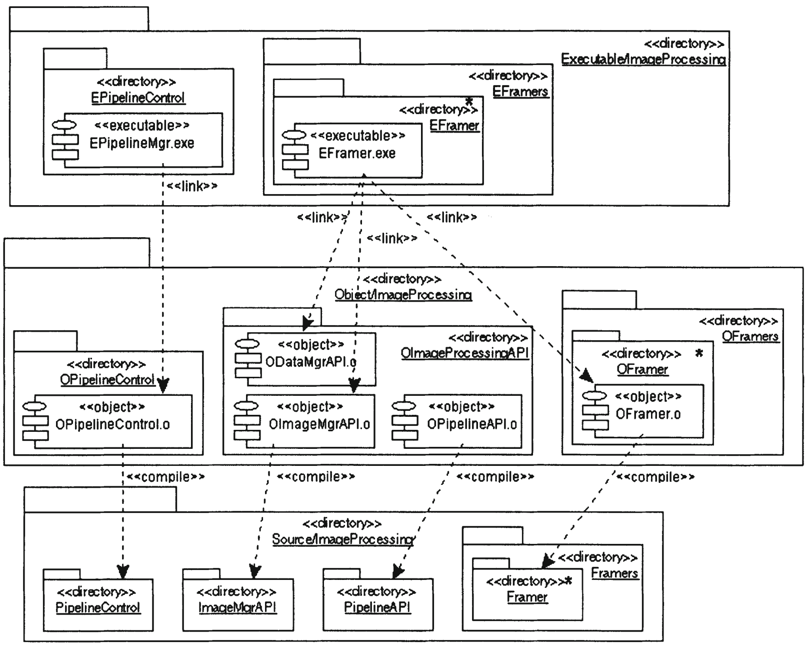
For the code view, we represent the source, object, and executable files as stereotyped classes, and the directories as stereotyped packages. The include, compile, and link relationships are shown as stereotyped dependencies. We use:

- Tables to describe the mapping between elements in the module and execution views and elements in the code view.
- UML Component Diagrams for showing the dependencies among source, intermediate, and executable files.

6. DISCUSSION

Table 8 summarizes the elements of our four architecture views and their corresponding UML Metamodel Classes and stereotype names, if any. For relations among the architecture description elements, we use UML associations and dependencies. We generally create a separate diagram for each kind of relation, but sometimes we combine them (e.g. the execution configuration diagram).

We use UML Class/Object, Package, and Component Diagrams for the elements and their relations, sometimes including the interfaces and attributes in these diagrams. Sequence Diagrams or State Diagrams are used to describe behavior.



<<directory>> ExecutableImageProcessing
<<directory>> EPipelineControl
<<executable>> EPipelineMgr.exe
<<directory>> EFramers
<<directory>> EFramer *
<<executable>> EFramer.exe
<<link>>
<<directory>> ObjectImageProcessing
<<directory>> OPipelineControl
<<object>> OPipelineControl.o
<<directory>> OImageProcessingAPI
<<object>> ODataMgrAPI.o
<<object>> OImageMgrAPI.o
<<object>> OPipelineAPI.o
<<directory>> OFramers
<<directory>> OFramer *
<<object>> OFramer.o
<<compile>>
<<directory>> SourceImageProcessing
<<directory>> PipelineControl
<<directory>> ImageMgrAPI
<<directory>> PipelineAPI
<<directory>> Framers
<<directory>> Framer *

Figure 8. Dependencies among source, object, and executable files

The configuration diagrams in the conceptual and execution views are UML Class/Object Diagrams, but we added some conventions to help define the semantics and improve the readability of the diagrams.

One convention is to use nesting to indicate decomposition. This makes the structure easier to see, although it can make layout difficult for complex structures. With this convention we cannot show recursive or indefinite nesting, which could be easily described in a diagram that depicts decomposition as a labeled association (a line) between two objects.

A semantic convention we use is that a configuration diagram describes the set of possible configurations at a single point in time. Systems generally have defined modes, e.g. start-up, shut-down, operational, diagnosis, recovery, etc. Each of these modes can have a different configuration, so should have a different diagram. In some modes (in our example, the operational mode) the configuration changes over time (in our case, pipelines are created and destroyed with each acquisition procedure). The dynamic behavior should be described separately. A sequence diagram works well to describe start-up and shut-down behavior.

Table 8. Summary of architecture description elements

Element	*UML Metamodel Class*	*Stereotype Name*
component	Class	<<component>>
port	Class	<<port>>
connector	Class	<<connector>>
role	label on association	
port or role protocol	Class	<<protocol>>
module	Class	<<module>>
subsystem	Package	<<subsystem>>
layer	Package	<<layer>>
run-time image	Class	<<process>>, <<shared data>>, <<thread>>, etc.
communication path	association	
source	Component	<<source>>
intermediate	Component	<<object>>
executable	Component	<<executable>>
directory	Package	<<directory>>

An important concern we have about using UML to describe software architecture is that the same notation can have a wide range of semantics. We use the same basic diagram, the UML Class/Object diagram to show most of the aspects of the architecture. We use stereotypes and special symbols to minimize the confusion between different views.

The more traditional use of UML is for the design of implementation classes for a system. We are also concerned that by using the same notation to describe the software architecture, we run the risk of further blurring the distinction between the architecture and the implementation. This is another reason to consistently use particular conventions, stereotypes, and special symbols for these architecture diagrams.

In summary, we found UML deficient in describing:

- correspondences: A graphical notation is too cumbersome for straightforward mappings such as the correspondence between elements in different views. This information is more efficiently described in a table (e.g.Table 3).
- protocols: The ability to show peer-to-peer communication is missing from UML. We used ROOM to describe protocols (e.g. Figure 2).
- ports on components: We used nesting to show the relationship between ports and components, but this is visually somewhat misleading. We would prefer a notation more similar to the lollipop notation for the interfaces of a module.
- dynamic aspects of the structure
- a general sequence of activities

UML worked well for describing:

- the static structure of the architecture
- variability: e.g. the conceptual configuration in Figure 1 describes the structure of a set of pipelines.
- a particular sequence of activities: e.g. the start-up behavior of an Image Pipeline (Figure 6).

REFERENCES

Bass, L., Clements, P., and Kazman, R. (1998) Software Architecture in Practice. Addison-Wesley, Massachusetts.

Eriksson, H., and Penker, M. (1998) UML Toolkit. John Wiley and Sons, London.

Fowler, M., with Scott, K. (1997) UML Distilled. Applying the Standard Object Modeling Language. Addison-Wesley, Massachusetts.

Hofmeister, C., Nord, R., Soni, D. (to appear) Applied Software Architecture. Addison-Wesley, Massachusetts.

Kramer, J., and Magee, J. (1990) The Evolving Philosophers Problem: Dynamic Change Management. *ACM Transactions on Software Engineering*, **16(11)**, 1293-1306.

Kruchten, P. (1995) The 4+1 View Model of Architecture, *IEEE Software*, **12(6)**.

Prieto-Diaz, R., and Neighbors, J.M. (1986) Module Interconnection Languages. *The Journal of Systems and Software*, **6(4)**, 307-334.

Purtilo, J.M. (1994) The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems,* **16(1)**, 151-174.

Selic, B., Gullekson, G., and Ward, P.T. (1994) Real-Time Object-Oriented Modeling. John Wiley and Sons, New York.

Selic, B., and Rumbaugh, J. (1998) Using UML for Modeling Complex Real-Time Systems. http://www.objectime.com/uml/uml.html.

Shaw, M., and Garlan, D. (1996) Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall.

Soni, D., Nord, R.L., and Hofmeister, C. (1995) Software Architecture in Industrial Applications, in Proceedings of the 17th International Conference on Software Engineering, Seattle, WA.

UML (1997) UML Notation Guide, Version 1.1. http://www.rational.com/uml.

Assessing the Suitability of a Standard Design Method for Modeling Software Architectures

Nenad Medvidovic and David S. Rosenblum
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, U.S.A.
{neno,dsr}@ics.uci.edu

Key words: Software architecture, architectural style, object-oriented design, architecture description languages, Unified Modeling Language

Abstract: Software architecture descriptions are high-level models of software systems. Most existing special-purpose architectural notations have a great deal of expressive power but are not well integrated with common development methods. Conversely, mainstream development methods are accessible to developers, but lack the semantics needed for extensive analysis. In our previous work, we described an approach to combining the advantages of these two ways of modeling architectures. While this approach suggested a practical strategy for bringing architectural modeling into wider use, it introduced specialized extensions to a standard modeling notation, which could also hamper wide adoption of the approach. This paper attempts to assess the suitability of a standard design method "as is" for modeling software architectures.

1. INTRODUCTION

Software architecture is an aspect of software engineering directed at reducing the costs of developing applications and increasing the potential for commonality among different members of a closely related product family (Garlan and Shaw, 1993; Perry and Wolf, 1992). Software development based on common architectural idioms has its focus shifted from lines-of-code to coarser-grained architectural elements and their overall interconnection structure. This enables developers to abstract away the unnecessary details and focus on the "big picture:" system structure, high level communication protocols, assignment of software components and connectors to hardware components, development process, and so forth (Garlan and Shaw, 1993; Kruchten, 1995; Luckham and Vera, 1995; Perry and Wolf, 1992; Soni, *et al.*, 1995; Taylor, *et al.*, 1996). The basic promise of software architecture research is that better software systems can result from modeling their important aspects during, and especially early in the development. Choosing which aspects to model and how to evaluate them are two decisions that frame software architecture research (Medvidovic and Rosenblum, 1997).

Part of the software architecture research community has focused on analytic evaluation of architecture descriptions. Many researchers have come to believe that, to obtain the benefits of an architectural focus, software architecture must be provided with its own body of specification languages and analysis techniques (Garlan, ed., 1995; Garlan, *et al.*, eds., 1995; Wolf, ed., 1996). Such languages are needed to demonstrate properties of a system upstream, thus minimizing the costs of errors. They are also needed to provide abstractions that are adequate for modeling a large system, while ensuring sufficient detail for establishing properties of interest. A large number of architecture description languages (ADLs) has been proposed (Allen and Garlan, 1997; Garlan, *et al.*, 1994; Luckham and Vera, 1995; Magee and Kramer, 1996; Medvidovic, Taylor, *et al.*, 1996; Moriconi, *et al.*, 1995; Shaw, DeLine, *et al.*, 1995; Vestal, 1996).

Each ADL embodies a particular approach to the specification and evolution of an architecture. Answering specific evaluation questions demands powerful, specialized modeling and analysis techniques that address specific aspects in depth. However, the emphasis on depth over breadth of the model can make it difficult to integrate these models with other development artifacts, because the rigor of formal methods draws the modeler's attention away from day-to-day development concerns. The use of special-purpose modeling languages has made this part of the architecture community fairly fragmented, as revealed by a recent survey of architecture description languages (Medvidovic and Taylor, 1997).

Another part of the community has focused on modeling a wide range of issues that arise in software development, perhaps with a family of models that span and relate the issues of concern. By paying the cost of making such models, developers gain the benefit of clarifying and communicating their understanding of the system. However, emphasizing breadth over depth potentially allows many problems and errors to go undetected, because lack of rigor allows developers to ignore certain details. Several competing notations have been used in this part of the community, but there now exists a concerted effort to standardize methods for object-oriented analysis and design (Object Management Group, 1996).

In our previous work, we described an approach to combining the advantages of specialized, highly formal methods of modeling architectures with general, less formal design methods (Robbins, *et al.*, 1998). This approach suggested a practical strategy for bringing architectural modeling into wider use, namely by incorporating substantial elements of architectural models into a standard design method, the Unified Modeling Language (UML) (Rational, 1997a). However, our technique is not without drawbacks: for each architectural approach and ADL, we introduced a somewhat specialized extension to UML. In particular, we relied heavily on UML's Object Constraint Language (OCL) (Rational and IBM, 1997) to specify architecture- and ADL-specific concepts.

OCL constraints are highly formal. Their formality may hamper wide adoption of our technique, although end users of the enhanced UML meta-model typically will not need to write OCL constraints. Furthermore, OCL is a part of the standard UML definition and it is expected that standardized UML tools will be able to process it. However, OCL is considered an uninterpreted part of UML and UML tools may not support it to the extent needed for creating, manipulating, analyzing, and evolving designs. For this reason, in this paper we attempt to assess the suitability of UML "as is" for modeling software architectures. In particular, we focus on one of the architectural approaches we addressed previously (Robbins, *et al.*, 1998), the C2 architectural style (Taylor, *et al.*, 1996). We use a simple meeting scheduler application to highlight the issues. In the process, we attempt to shed light on the relationship between architecture and design.

The paper is organized as follows. The next section briefly describes UML. *Section 3* briefly describes the example application, a meeting scheduler, used to illustrate our arguments throughout the paper. In *Section 4*, we introduce the C2 style and discuss a possible C2 architecture for the meeting scheduler application. *Section 5* provides a "C2 style" UML design of the meeting scheduler. We discuss the results and key lessons learned in *Section 6*. Our conclusions round out the paper.

2. OVERVIEW OF UML

2.1 UML background

A UML model of a software system consists of several partial models, each of which addresses a certain set of issues at a certain level of fidelity. There are eight issues addressed by UML models:

1. classes and their declared attributes, operations, and relationships;
2. the possible states and behavior of individual classes;
3. packages of classes and their dependencies;
4. example scenarios of system usage including kinds of users and relationships between user tasks;
5. the behavior of the overall system in the context of a usage scenario;
6. examples of object instances with actual attributes and relationships in the context of a scenario
7. examples of the actual behavior of interacting instances in the context of a scenario; and
8. the deployment and communication of software components on distributed hosts.

Fidelity refers to how close the model will be to the eventual implementation of the system: low-fidelity models tend to be used early in the life-cycle and are more problem-oriented and generic, whereas high-fidelity models tend to be used later and are more solution-oriented and specific. Increasing fidelity demands effort and knowledge to build more detailed models, but results in more properties of the model holding true in the system.

UML is a graphical language with fairly well-defined syntax and semantics. The syntax of the graphical presentation is specified by examples and a mapping from graphical elements to elements of the underlying semantic model (Rational, 1997c). The syntax and semantics of the underlying model are specified semi-formally via a meta-model, descriptive text, and constraints (Rational, 1997b). The meta-model is itself a UML model that specifies the abstract syntax of UML models. This is much like using a BNF grammar to specify the syntax of a programming language. For example, the UML meta-model states that a Class is one kind of model element with certain attributes, and that a Feature is another kind of model element with its own attributes, and that there is a one-to-many composition relationship between them.

UML is an extensible language in that new constructs may be added to address new issues in software development. Three mechanisms are provided to allow limited extension to new issues without changing the existing syntax or semantics of the language. (1) *Constraints* place semantic restrictions on particular design elements. (2) *Tagged values* allow new attributes to be added to particular elements of the model. (3) *Stereotypes* allow groups of constraints and tagged values to be given descriptive names and applied to other model elements; the semantic effect is as if the constraints and tagged values were applied directly to those elements. Another possible extension mechanism is to modify the meta-model, but this approach results in a completely new notation to which standard UML tools cannot be applied. We discuss this approach in more detail in *Section 2.2*. *Figure 1* shows the parts of the UML meta-model used in this paper. We have simplified the meta-model for purposes of illustration.

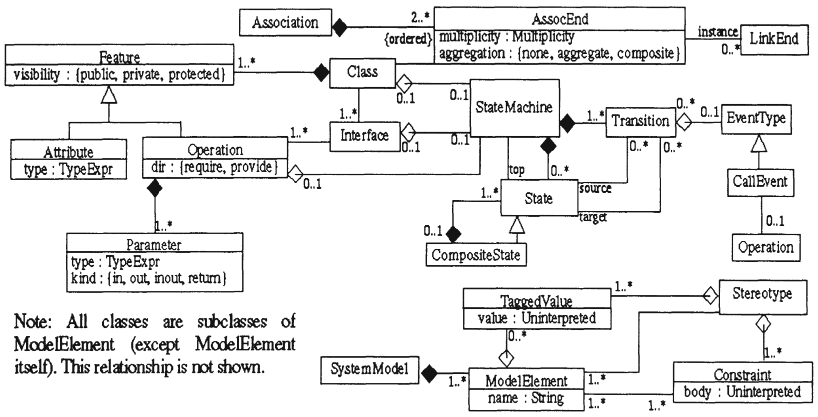


Figure 1. Simplified UML meta-model, adapted from (Rational, 1997b).

2.2 Our strategy for adapting UML for architecture modeling

In (Robbins, *et al.*, 1998) we studied two possible approaches to using UML to model architectures. One approach is to define an ADL-specific meta-model. This approach has been used in more comprehensive formalizations of architectural styles (Abowd, *et al.*, 1995; Medvidovic, Taylor, *et al.*, 1996). Defining a new meta-model helps to formalize the ADL, but does not aid integration with standard design methods. By defining our new meta-classes as subclasses of existing meta-classes we would achieve some integration. For example, defining Component as a subclass of meta-class Class would give it the ability to participate in any relationship in which Class can participate. This is basically the integration that we desire. However, this integration approach requires *modifications* to the meta-model that would not *conform* to the UML standard; therefore, we cannot expect UML-compliant tools to support it.

The approach for which we opted instead was to restrict ourselves to using UML's built-in extension mechanisms on existing meta-classes (Robbins, *et al.*, 1998). This allows the use of existing and future UML-compliant tools to represent the desired architectural models, and to support architectural style conformance checking when OCL-compliant tools become available. Our basic strategy was to first choose an existing meta-class from the UML meta-model that is semantically close to an ADL construct, and then define a stereotype that can be applied to instances of that meta-class to constrain its semantics to that of the ADL.

Neither of the two approaches answers the deeper question of UML's suitability for modeling software architectures "as is," i.e., without defining meta-models specific to a particular architectural approach or extending the existing UML meta-model. Such an exercise would highlight the respective advantages of special- and general-purpose design notations in modeling architectures. It also has the potential to further clarify the relationship between software architecture and design. Therefore, in this paper we study the characteristics of using the existing UML features to model architectures in a particular style, C2.

3. EXAMPLE APPLICATION

The example we selected to motivate the discussion in this paper is a simplified version of the meeting scheduler problem, initially proposed by (van Lamsweerde, *et al.*, 1992)and recently considered as a candidate model problem in software architectures (Shaw, Garlan, *et al.*, 1995). We have chosen this problem partly because of our prior experience with designing and implementing a distributed meeting scheduler in the C2 architectural style, described in (Taylor, *et al.*, 1996).

Meetings are typically arranged in the following way. A meeting initiator asks all potential meeting attendees for a set of dates on which they cannot attend the meeting (their "exclusion" set) and a set of dates on which they would prefer the meeting to take place (their "preference" set). The exclusion and preference sets are contained in some time interval prescribed by the meeting initiator (the "date range").

The initiator also asks active participants to provide any special equipment requirements on the meeting location (e.g., overhead-projector, workstation, network connection, telephones); the initiator may also ask important participants to state preferences for the meeting location.

The proposed meeting date should belong to the stated date range and to none of the exclusion sets. It should also ideally belong to as many preference sets as possible. A date conflict occurs when no such date can be found. A conflict is strong when no date can be found within the date range and outside all exclusion sets; it is weak when dates can be found within the date range and outside all exclusion sets, but no date can be found at the intersection of all preference sets. Conflicts can be resolved in several ways:

– the initiator extends the date range;

- some participants expand their preference set or narrow down their exclusion set; or
- some participants withdraw from the meeting.

4. MODELING THE EXAMPLE APPLICATION IN C2

4.1 Overview of C2

C2 is a software architectural style for user interface intensive systems (Taylor, *et al.*, 1996). C2SADEL is an ADL for describing C2-style architectures (Medvidovic, Taylor, *et al.*, 1996; Medvidovic, Oreizy, *et al.*, 1996); henceforth, in the interest of clarity, we use "C2" to refer to the combination C2 and C2SADEL. In a C2-style architecture, *connectors* transmit messages between components, while *components* maintain state, perform operations, and exchange messages with other components via two interfaces (named "top" and "bottom"). Each interface consists of a set of messages that may be sent and a set of messages that may be received. Inter-component messages are either *requests* for a component to perform an operation, or *notifications* that a given component has performed an operation or changed state.

In the C2 style, components may not directly exchange messages; they may only do so via connectors. Each component interface may be attached to at most one connector. A connector may be attached to any number of other components and connectors. Request messages may only be sent "upward" through the architecture, and notification messages may only be sent "downward."

The C2 style further demands that components communicate with each other only through message-passing, never through shared memory. Also, C2 requires that notifications sent from a component correspond to the operations of its internal object, rather than the needs of any components that receive those notifications. This constraint on notifications helps to ensure *substrate independence*, which is the ability to reuse a C2 component in architectures with differing substrate components (e.g., different window systems). The C2 style explicitly does not make any assumptions about the language(s) in which the components or connectors are implemented, whether or not components execute in their own threads of control, the deployment of components to hosts, or the communication protocol(s) used by connectors.

4.2 Modeling the meeting scheduler in C2

Figure 2 shows a graphical depiction of a possible C2-style architecture for a simple meeting scheduler system. This system consists of components supporting the functionality of a *MeetingInitiator* and several potential meeting *Attendees* and *ImportantAttendees*. Three C2 connectors are used to route messages among the components. Certain messages from the *Initiator* are sent both to *Attendees* and *ImportantAttendees*, while others (e.g., to obtain meeting location preferences) are only routed to *ImportantAttendees*. Since a C2 component has only one communication port on its top and one on its bottom, and all message routing functionality is relegated to connectors, it is the responsibility of *MainConn* to ensure that *AttConn* and *ImportantAttConn* above it receive only those message relevant to their respective attached components.

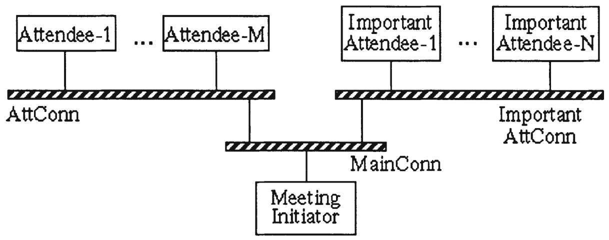


Figure 2. A C2-style architecture for a meeting scheduler system.

The *Initiator* component sends requests for meeting information to *Attendees* and *ImportantAttendees*. The two sets of components notify the *Initiator* component, which attempts to schedule a meeting and either requests that each potential attendee mark it in his/her calendar (if the meeting can be scheduled), or it sends other requests to attendees to extend the date range, remove a set of excluded dates, add preferred dates, or withdraw from the meeting. Each *Attendee* and *ImportantAttendee* component, in turn, notifies the *Initiator* of its date, equipment, and location preferences, as well as excluded dates. *Attendee* and *ImportantAttendee* components cannot make requests of the *MeetingInitiator* component, since they are above it in the architecture.

Most of this information is implicit in the graphical view of the architecture shown in *Figure 2*. For this reason, we specify the architecture in C2SADEL, a textual language for modeling C2-style architectures (Medvidovic, 1996; Medvidovic, Taylor, *et al.*, 1996; Medvidovic, Oreizy, *et al.*, 1996). For simplicity, we assume that all attendees' equipment needs will be met, and that a meeting location will be available on the given date and that it will be satisfactory for all (or most) of the important attendees.

The *MeetingInitiator* component is specified below. The component only communicates with other parts of the architecture through its top port.

```
component MeetingInitiator is
    interface
        top_domain is
            out
                GetPrefSet ();
                GetExclSet ();
                GetEquipReqts ();
                GetLocPrefs ();
                RemoveExclSet ();
                RequestWithdrawal (to Attendee);
                RequestWithdrawal (to ImportantAttendee);
                AddPrefDates ();
                MarkMtg (d : date; l : loc_type);
            in
                PrefSet (p : date_rng);
                ExclSet (e : date_rng);
                EquipReqts (eq : equip_type);
                LocPref (l : loc_type);
        bottom_domain is
            out null;
            in null;
    parameters null;

    methods
        procedure Start ();
        procedure Finish ();
        procedure SchedMtg (p : set date_rng; e : set date_rng);
        procedure AddPrefSet (pref : date_rng);
        procedure AddExclSet (exc : date_rng);
        procedure AddEquipReqts (eq : equip_type);
        procedure AddLocPref (l : loc_type);
        function AttendInfoCompl () return boolean;
        procedure IncNumAttends (n : integer);
        function GetNumAttends () : return integer;

    behavior
        startup
            invoke_methods Start;
            always_generate GetPrefSet, GetExclSet, GetEquipReqts,
                            GetLocPrefs;
        cleanup
            invoke_methods Finish;
            always_generate null;
        received_messages PrefSet;
            invoke_methods AddPrefSet, IncNumAttends, AttendInfoCompl,
                           GetNumAttends, SchedMtg;
            may_generate RemoveExclSet xor RequestWithdrawal xor
                         MarkMtg;
        received_messages ExclSet;
            invoke_methods AddExclSet, AttendInfoCompl, GetNumAttends,
                                SchedMtg;
                    may_generate AddPrefDates xor RemoveExclSet xor
                              RequestWithdrawal xor MarkMtg;
               received_messages EquipReqts;
                    invoke_methods AddEquipReqts, AttendInfoCompl,
                                GetNumAttends, SchedMtg;
                    may_generate AddPrefDates xor RemoveExclSet xor
                              RequestWithdrawal xor MarkMtg;
               received_messages LocPref;
                    invoke_methods AddLocPref;
                    always_generate null;

          context
               bottom_most computational_unit;
     end MeetingInitiator;
```

The *Attendee* and *ImportantAttendee* components receive meeting scheduling requests from the *Initiator* and notify it of the appropriate information. The two types of components only communicate with other parts of the architecture through their bottom ports.

```
     component Attendee is
          interface
               top_domain is
                    out null;
                    in null;
               bottom_domain is
                    out
                         PrefSet (p : date_rng);
                         ExclSet (e : date_rng);
                         EquipReqts (eq : equip_type);
                         Witdrawn ();
                    in
                         GetPrefSet ();
                         GetExclSet ();
                         GetEquipReqts ();
                         RemoveExclSet ();
                         RequestWithdrawal ();
                         AddPrefDates ();
                         MarkMtg (d : date; l : loc_type);
          parameters null;

          methods
               procedure Start ();
               procedure Finish ();
               procedure NoteMtg (d : date; l : loc_type);
               function DeterminePrefSet () return date_rng;
               function DetermineExclSet () return date_rng;
               function AddPrefDates () return date_rng;
               function RemoveExclSet () return date_rng;
               procedure DetermineEquipReqts (eq : equip_type);

          behavior
               startup
                    invoke_methods Start;
                    always_generate null;
                cleanup
                    invoke_methods Finish;
                    always_generate null;
                received_messages GetPrefSet;
                    invoke_methods DeterminePrefSet;
                    always_generate PrefSet;
                received_messages AddPrefDates;
                    invoke_methods AddPrefDates;
                    always_generate PrefSet;
                received_messages GetExclSet;
                    invoke_methods DetermineExclSet;
                    always_generate ExclSet;
                received_messages GetEquipReqts;
                    invoke_methods DetermineEquipReqts;
                    always_generate EquipReqts;
                received_messages RemoveExclSet;
                    invoke_methods RemoveExclSet;
                    always_generate ExclSet;
                received_messages RequestWithdrawal;
                    invoke_methods Finish;
                    always_generate Withdrawn;
                received_messages MarkMtg;
                    invoke_methods NoteMtg;
                    always_generate null;

        context
            top_most computational_unit;
    end Attendee;
```

ImportantAttendee is a specialization of the *Attendee* component: it duplicates all of *Attendee*'s functionality and adds specification of meeting location preferences. *ImportantAttendee* is thus specified as a subtype of *Attendee* that preserves its interface and behavior, but can implement that behavior in a new manner.

```
    component ImportantAttendee is subtype Attendee (int and beh)
        interface
            bottom_domain is
                out
                    LocPrefs (l : loc_type);
                in
                    GetLocPrefs ();
        methods
            function DetermineLocPrefs () return loc_type;
        behavior
            received_messages GetLocPrefs;
                invoke_methods DetermineLocPrefs;
                always_generate LocPrefs;
    end ImportantAttendee;
```

The *MeetingScheduler* architecture depicted in *Figure 2* is shown below. The architecture is specified with conceptual components (i.e., component types). Each conceptual component (e.g., *Attendee*) can be instantiated multiple times in a system.

```
architecture MeetingScheduler is
    conceptual_components
        top_most
            Attendee;
            ImportantAttendee;
        internal null;
        bottom_most
            MeetingInitiator;
    connectors
        connector MainConn is
            message_filter no_filtering;
        end MainConn;
        connector AttConn is
            message_filter no_filtering;
        end AttConn;
        connector ImportantAttConn is
            message_filter no_filtering;
        end ImportantAttConn;
    architectural_topology
        connector AttConn connections
            top_ports
                Attendee;
            bottom_ports
                MainConn;
        connector ImportantAttConn connections
            top_ports
                ImportantAttendee;
            bottom_ports
                MainConn;
        connector MainConn connections
            top_ports
                AttConn;
                ImportantAttConn;
            bottom_ports
                MeetingInitiator;
end MeetingScheduler;
```

An instance of the architecture (a system) is specified by instantiating the components. For example, an instance of the meeting scheduler application with three participants and two important participants is specified as follows.

```
system MeetingScheduler_1 is
    architecture MeetingScheduler with
            Attendee instance Att_1, Att_2, Att_3;
            ImportantAttendee instance ImpAtt_1, ImpAtt_2;
            MeetingInitiator instance MtgInit_1;
end MeetingScheduler_1;
```

5. MODELING THE C2-STYLE MEETING SCHEDULER APPLICATION IN UML

The process of designing a C2-style application in UML should be driven and constrained both by the rules of C2 and the modeling features available in UML. The two must be considered simultaneously. For this reason, the initial steps in this process are to develop a domain model for a given application in UML and an informal C2 architectural diagram, such as the one from *Figure 2*. Such an architectural diagram is key to making the appropriate mappings between classes in the domain and architectural components. Furthermore, it points to the need to explicitly model connectors in any C2-style architecture. Another important aspect of C2 architectures is the prominence of components' message interfaces. This is reflected in a UML design by modeling interfaces explicitly and independently of the classes that will implement those interfaces.

Our initial attempt at a UML class diagram for the meeting scheduler application is shown in *Figure 3*. The diagram shows the domain model for the meeting scheduler application consisting of the domain classes, their inheritance relationships, and their associations.

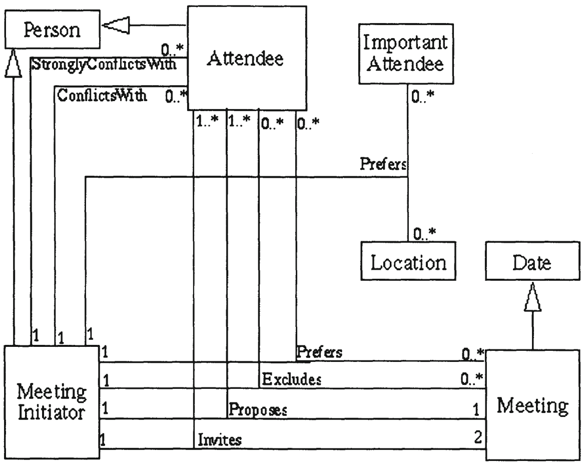


Figure 3. UML class diagram for the meeting scheduler application. Details (attributes and methods) of each individual class have been suppressed for clarity.

The diagram abstracts away many architectural details, such as the mapping of classes in the domain to implementation components, the order of interactions among the different classes, and so forth. Furthermore, much of the semantics of class interaction is missing from the diagram. For example, the *Invites* association associates two *Meetings* with one or more *Attendees* and one *MeetingInitiator*. However, the association does not make clear the fact that the two *Meetings* are intended to represent a range of possible meeting dates, rather than a pair of related meetings.

Each class exports one or more interfaces, shown in *Figure 4*. The *ImportantMtgInit* and *ImportantMtgAttend* interfaces inherit from the *MtgInit* and *MtgAttend* interfaces, respectively. The only difference is the added operation to request and notify of location preferences.

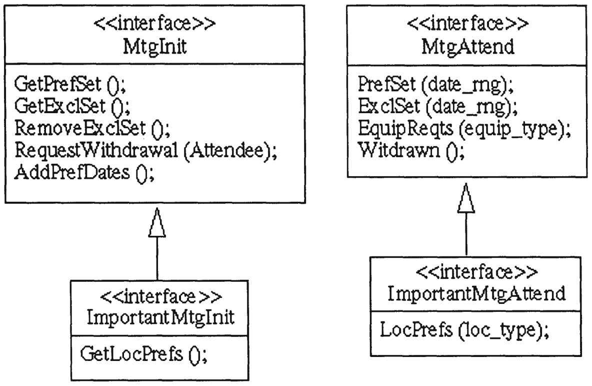


Figure 4. Meeting scheduler class interfaces.

Note that every interface element corresponds to a C2 message in the architecture specified in *Section 4.2*. All methods in the UML design will be implemented as asynchronous message passes, as they would in C2. Since C2 components communicate via implicit invocation, C2 messages do not have return values; this is also reflected in *Figure 4*.

In order to model a C2 architecture in UML, connectors must be defined. Although connectors fulfill a role different from components, they can also be modeled with UML classes. However a C2 connector is by definition generic and can accommodate any number ant type of C2 components; informally, the interface of a C2 connector is a union of the interfaces of its attached components. UML does not support this form of genericity, so that the connectors specified in UML have to be application-specific. For that purpose, the connectors for the meeting scheduler application share the components' interfaces. Each connector can be thought of as a simple class that forwards each message it receives to the appropriate components. Therefore, while the component class interface specifications, shown in *Figure 4*, correspond to the different C2 components' outgoing messages (i.e., their provided functionality), the connector interfaces are routers of both the incoming and outgoing messages, as depicted in *Figure 5*. Connectors do not add any functionality at the domain model level; we have thus chosen to omit them from the class diagram in *Figure 3*.

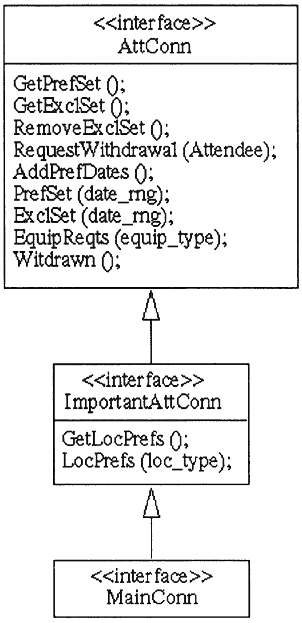


Figure 5. Application-specific UML classes representing C2 connectors.

A refined class diagram for the meeting scheduler application is shown in *Figure 6*. The *Attendee* and *ImportantAttendee* classes are related by interface inheritance, which is depicted in *Figure 4*, but is only implicit in *Figure 6* (and altogether omitted from *Figure 3*). We have omitted from *Figure 6* the *Location*, *Meeting*, and *Date* classes shown in *Figure 3*, since they have not been impacted. We have also omitted the two superclasses for the components and connectors (*Person* and *Conn*, respectively).

Note that the class diagram in *Figure 6* is similar in its structure to the C2 architecture depicted in *Figure 2*. The only difference is that the diagram in *Figure 2* depicts instances of the different components and connectors, while a UML class diagram depicts classes and their associations. UML provides several types of diagrams that depict class instances (objects). One candidate is UML's object diagrams; however, we choose to depict a collaboration diagram to further draw the contrast between UML and C2.

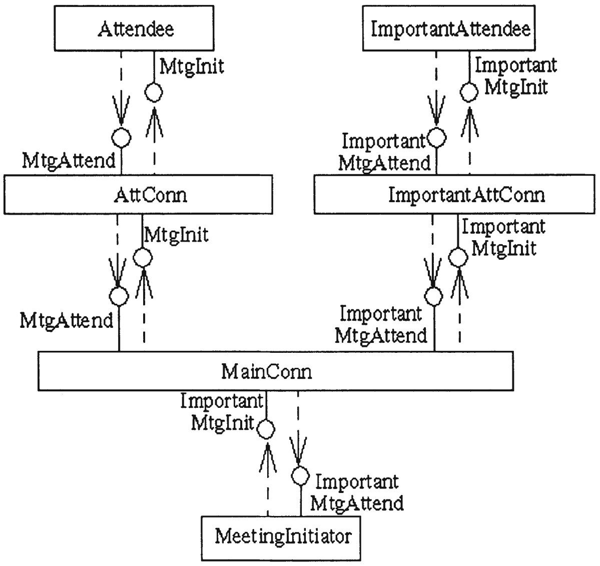


Figure 6. UML class diagram for the meeting scheduler application designed in the C2 architectural style.

Figure 7 shows the collaboration between an instance of the *MeetingInitiator* class (*MI*) and any instances of *Attendee* and *ImportantAttendee* classes: *MI* issues a request for a set of preferred meeting dates; *MC*, an instance of the *MainConn* class routes the request to instances of both connectors above it, *AC* and *IAC*, which, in turn, route the requests to all components attached on their top sides; each participant component chooses a preferred date and notifies any components below it of that choice; these notification messages will eventually be routed to *MI* via the connectors. Note that, if *MI* had sent the request to get meeting location preferences (*GetLocPrefs* in the *ImportantMtgInit* interface in *Figure 4*), *MC* would have routed them only to *IAC* and none of the instances of the *Attendee* class would have received that request

The diagrams in this section, and particularly *Figure 6*, differ from a C2 architecture in that they explicitly specify only the messages a component receives (via interface attachments to a component rectangle). UML also allows specification of messages a component sends; we believe those messages to be obvious from the diagram and have thus chosen to omit them to simplify the diagrams.

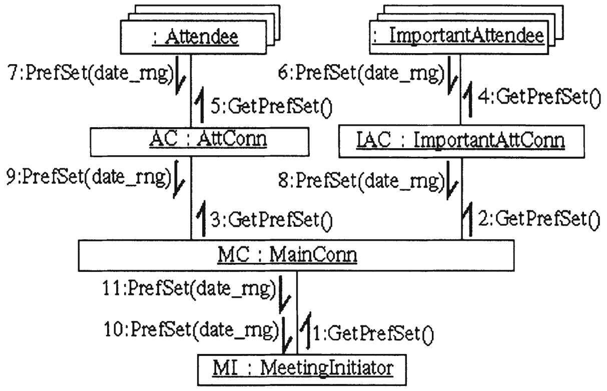


Figure 7. Collaboration diagram for the meeting scheduler application showing a response to a request issued by the *MeetingInitiator* to both *Attendees* and *ImportantAttendees*.

6. DISCUSSION

The exercise of modeling a C2-style architecture in UML has been fairly successful. Part of the success can be attributed to the fact that many architectural concepts are found in UML (e.g., interfaces, component associations, behavioral modeling, and so forth). On the other hand, the modeling capabilities provided by UML do not always fully satisfy the needs of architectural description. We discuss several major similarities and differences in this section.

6.1 Software modeling philosophies

Neither C2 nor UML constrain the choice of implementation language or require that any two components be implemented in the same language or thread of control. C2 limits communication to asynchronous message passing and UML supports this restriction. Both C2 and UML include specifications of messages that may be sent and received.

Although we did not model details of the internal parts of a C2 component or the behavior of any C2 constructs (components, connectors, communication ports, and so forth) in our UML specification, we believe that many of those aspects could be modeled with UML's sequence, collaboration, statechart, and activity diagrams. Existing ADLs, including C2SADEL, are often not able to support all of these kinds of semantic models (Medvidovic and Taylor, 1997).

6.2 Assumptions

Like any notation, UML embodies its developers' assumptions about its intended usage. "Architecting" a system was not an intended use of UML. While one can indeed focus on the different perspectives when modeling a system (discussed above), a software architect may find that the support for those perspectives found in UML only partially satisfies his/her needs.

For example, in modeling the collaboration among C2 components shown in *Figure 7*, we were forced to assign a relative ordering to messages in the architecture. In effect, since all C2 components and connectors can execute in their own thread(s) of control, such an ordering cannot always be determined. Indeed, it is possible that message 4 would be sent before message 3.

6.3 Problem domain modeling

UML supports modeling a problem domain, as we have briefly shown in this paper. A C2 architectural model, however, often hides some of the information present in a domain model. For example, meeting, equipment, and location information is present in *Figure 3*, but is missing from the C2 architecture specified in *Section 4* and its corresponding UML diagram in *Figure 6*. Modeling all the relevant information early in the development lifecycle is crucial to the success of a software project. Therefore, a domain model should be considered a separate and useful architectural perspective (Medvidovic and Rosenblum, 1997; Tracz, 1995).

6.4 Architectural abstractions

Some concepts of C2, and software architectures in general, are very different from those of UML and object-oriented design in general. Connectors are first-class entities in C2. While the functionality of a connector can typically be abstracted by a class/component (Luckham and Vera, 1995; Magee and Kramer, 1996), C2 connectors have the added property that their interfaces are context-reflective. This property is designed into C2SADEL and C2's implementation infrastructure (Medvidovic, *et al.*, 1997) for all connectors, whereas the approach described in this paper requires specialized modeling of application-specific connector classes in UML.

The underlying problem is even deeper. Although UML may provide modeling power equivalent to or surpassing that of an ADL, the abstractions it provides may not match an architect's mental model of the system as faithfully as the architect's ADL of choice. If the primary purpose of a language is to provide a vehicle of expression that matches the intuitions and practices of users, then that language should aspire to reflect those intentions and practices (Shaw and Garlan, 1995). We believe this to be a key issue and one that argues against considering a notation like UML to be a "mainstream" ADL: a given language (e.g., UML) offers a set of abstractions that an architect uses as design tools; if certain abstractions (e.g., components and connectors) are buried in others (e.g., classes), the architect's job is made more (and unnecessarily) difficult; separating components from connectors, raising them both to visibility as top-level abstractions, and endowing them with certain features and limitations also raises them in the consciousness of the designer.

6.5 Architectural styles

Architecture is the appropriate level of abstraction at which rules of a compositional style (i.e., an architectural style) can be exploited and should be elaborated. Doing so results in a set of heuristics that, if followed, will guarantee a resulting system certain desirable properties.

Standard UML provides no support for architectural styles. The rules of different styles have to be built into UML by constraining its meta-model, as we have done previously (Robbins, *et al.*, 1998)]. Therefore, in choosing to use UML "as is", we have removed one shortcoming of our previous approach, only to introduce another. In particular, every C2 architecture designed in the manner we described in this paper adheres to the UML meta-model and, as such, can be understood by a typical UML user and manipulated with standardized UML tools. On the other hand, the process of modeling a C2 architecture in UML is heuristic- rather than constraint-driven. Therefore, there is no guarantee that the designer will always adhere to the rules of C2. For this reason, it may also be more difficult to provide support for automated translation of "C2-style" UML designs into C2SADEL for C2-specific manipulations.

7. CONCLUSIONS

We found this initial attempt at modeling a C2-style architecture in UML useful. It highlighted those UML characteristics that show potential for aiding architectural modeling, but also pointed out some of UML's shortcomings in this regard. This experience can also serve as a solid basis for further study, both with other C2 architectures, as well as with other ADLs, e.g., Wright (Allen and Garlan, 1997), and architectural styles, e.g., client-server.

Before we can draw definitive conclusions about the relative merits of this approach and the approach described in our previous work (Robbins, *et al.*, 1998), further research into the techniques described in the two papers is needed. One necessary step to integrate UML with other ADLs discussed in (Robbins, *et al.*, 1998): Wright (Allen and Garlan, 1997), Darwin (Magee and Kramer, 1996), and Rapide (Luckham and Vera, 1995). Each of these ADLs has certain aspects in common with UML; these were expressed with UML's extension mechanisms. We intend to investigate whether they can also be expressed in UML without extensions.

Our experience to date indicates that adapting UML to address architectural concerns requires reasonable effort, has the potential to be a useful complement to ADLs and their analysis tools, and may be a practical step toward mainstream architectural modeling. Using UML has the benefits of leveraging mainstream tools, skills, and processes. It may also aid in the comparison of ADLs because it forces some implicit assumptions to be explicitly stated in common terms.

ACKNOWLEDGEMENTS

We wish to thank J. Robbins and D. Redmiles for their insights into the issues in integrating UML with architectures and their collaboration on other aspects of this work.

Effort sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021 and by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061. This material is also partially based on work supported by the National Science Foundation under Grant No. CCR-9701973. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, Air Force Office of Scientific Research or the U.S. Government.

REFERENCES

Abowd, G.; Allen, R. and Garlan, D. (1995), Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, pp. 319-364 (October).

Allen, R. and Garlan, D. (1997), A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, pp. 213-249 (July).

Garlan, D.; editor (1995). *Proceedings of the First International Workshop on Architectures for Software Systems*, Seattle, WA (April).

Garlan, D.; Allen, R. and Ockerbloom, J. (1994), Exploiting Style in Architectural Design Environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, pp. 175-188, New Orleans, Louisiana, USA (December).

Garlan, D.; Paulisch, F. N. and Tichy, W. F.; editors (1995), Summary of the Dagstuhl Workshop on Software Architecture, February 1995. Reprinted in ACM Software Engineering Notes, pp. 63-83 (July).

Garlan, D. and Shaw, M. (1993), *An introduction to software architecture: Advances in software engineering and knowledge engineering*, volume I. World Scientific Publishing.

Kruchten, P. B. The 4+1 view model of architecture. *IEEE Software*, pp. 42-50, November 1995.

van Lamsweerde, A.; Darimont, R. and Massonet, P. (1992), The Meeting Scheduler System: Preliminary Definition. University of Louvain, Unite d'informatique, B-1348 Louvain-la-Neuve, Belgium (October).

Luckham, D. C. and Vera, J. (1995), An event-based architecture definition language. *IEEE Transactions on Software Engineering*, pp. 717-734 (September).

Magee, J. and Kramer, J. (1996), Dynamic structures in software architecture. *In Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pp. 3-14, San Francisco, CA (October).

Medvidovic, N. (1996), ADLs and Dynamic Architecture Changes. In A. L. Wolf, ed., *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pp. 24-27, San Francisco, CA (October).

Medvidovic, N.; Taylor, R. N. and Whitehead, E. J., Jr. (1996), Formal Modeling of Software Architectures at Multiple Levels of Abstraction. In *Proceedings of the California Software Symposium 1996*, pp. 28-40, Los Angeles, CA (April).

Medvidovic, N. and Rosenblum, D. S. (1997), Domains of Concern in Software Architectures and Architecture Description Languages. In *Proceedings of the USENIX Conference on Domain Specific Languages*, pp. 199-212, Santa Barbara, CA (October).

Medvidovic, N. and Taylor, R. N. (1997), A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Sixth European Software Engineering Conference together with Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 60-76, Zurich, Switzerland (September).

Medvidovic, N.; Oreizy, P.; Robbins, J. E. and Taylor, R. N. (1996), Using object-oriented typing to support architectural design in the C2 style. In *Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pp. 24-32, San Francisco, CA (October).

Medvidovic, N.; Oreizy, P. and Taylor, R. N. (1997), Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)*, pp. 190-198, Boston, MA (May). Also in *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, pp. 692-700, Boston, MA (May).

Moriconi, M.; Qian, X. and Riemenschneider, R. A. (1995), Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, pp. 356-372 (April).

Object Management Group (1996), Object analysis and design RFP-1. Object Management Group document ad/96-05-01 (June). Available from http://www.omg.org/docs/ad/96-05-01.pdf.

Perry, D. E. and Wolf, A. L. (1992), Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, pp. 40-52 (October).

Rational Partners (Rational, IBM, HP, Unisys, MCI, Microsoft, ObjecTime, Oracle, i-Logix, etc.) (1997a), Proposal to the OMG in response to OA&D RFP-1. Object Management Group document ad/97-07-03 (July). Available from http://www.omg.org/docs/ad/.

Rational Partners (1997b), UML Semantics. Object Management Group document ad/97-08-04 (September). Available from http://www.omg.org/docs/ad/97-08-04.pdf.

Rational Partners (1997c), UML Notation Guide. Object Management Group document ad/97-08-05 (September). Available from http://www.omg.org/docs/ad/97-08-05.pdf.

Rational Software Corporation and IBM (1997), Object constraint language specification. Object Management Group document ad/97-08-08 (September). Available from http://www.omg.org/docs/ad/.

Robbins, J. E.; Medvidovic, N.; Redmiles, D. F. and Rosenblum, D. S. (1998), Integrating Architecture Description Languages with a Standard Design Method. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pp. 209-218, Kyoto, Japan (April).

Shaw, M.; DeLine, R.; Klein, D. V.; Ross, T. L.; Young, D. M. and Zelesnik, G. (1995), Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, pp. 314-335 (April).

Shaw M. and Garlan D. (1995), Formulations and Formalisms in Software Architecture. Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, Springer-Verlag Lecture Notes in Computer Science, Volume 1000.

Shaw, M.; Garlan, D.; Allen, R.; Klein, D.; Ockerbloom, J.; Scott, C. and Schumacher, M. (1995), Candidate Model Problems in Software Architecture. Unpublished manuscript (November). Available from http://www.cs.cmu.edu/afs/cs/project/compose/www/html/ModProb/.

Soni, D.; Nord, R. and Hofmeister, C. (1995), Software Architecture in Industrial Applications. In *Proceedings of the 17th International Conference on Software Engineering*, pp. 196-207, Seattle, WA (April).

Taylor, R. N.; Medvidovic, N.; Anderson, K. M.; Whitehead, E. J., Jr.; Robbins, J. E.; Nies, K. A.; Oreizy, P. and Dubrow, D. L. (1996), A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, pp. 390-406 (June).

Tracz, W. (1995), DSSA (Domain-Specific Software Architecture) Pedagogical Example. *ACM SIGSOFT Software Engineering Notes* (July).

Vestal, S. (1996), MetaH Programmer's Manual, Version 1.09. Technical Report, Honeywell Technology Center (April).

Wolf, A. L.; editor (1996), *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, San Francisco, CA (October).

Software Architecture and Java Beans

Sylvia Stuurman
Delft University of Technology
Department of Computer Science,
P.O. Box 356, 2600AJ The Netherlands
S.Stuurman@twi.tudelft.nl

Key words: Java Beans, software architecture, component based development

Abstract: In theory, software architecture and component-based development make an ideal match: the concerns of software architecture are high level design, interaction, and configuration of components, while component-based development is centered on the implementation and specification of reusable components.
Together, these concerns seem to be the yin and yang for the development of complex systems out of existing components. However, several authors have already explained that in reality, there is a gap between the two areas. In this paper, we investigate the relation between Java Beans and a software architecture description: may Java Beans simply be used as ready-to-use implementations of a software architecture? What restrictions do they impose on the software architecture? Where are the mismatches?

1. INTRODUCTION

As has been argued many times, today's complex large-scale software systems ask for a different kind of software engineering than small and simple programs. On the one hand, there is a need for very high-level design. The level of abstraction should be higher than that of objects, or procedures. Moreover, such a design should be a model of the system-in-use, not just a model of the implementation (Allen and Garlan, 1994). Software architecture is an answer for this need (we use "software architecture" in the sense of the definition of Garlan and Shaw (Garlan and Shaw, 1993):

"Structural issues include gross organisation and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.").

On the other hand, there is a need for the reuse of components. The ideal here is, for a developer, to be able to shop among different component providers, and build a system in the same way as building a vehicle out of Lego bricks and pieces. Several standards, both commercial and non-commercial, for component models have arisen, such as CORBA, ActiveX, and Java Beans.

Software architecture seems a natural complement for reusable software components: existing component middleware technologies are component-centric, and they standardize external component properties. Software architectures are system-centric, with more emphasis on the connections, and the properties of the system as a whole.

One of the problems of building systems out of existing components is the possibility of an "architectural mismatch" between components (Garlan et al., 1995). Components make implicit assumptions about the nature of the components (infrastructure, control model, data model), the nature of the connections (protocols and data model) and about the global architectural structure (for instance about the presence or absence of particular components or connections). As part of a solution for this problem, it has been suggested (Garlan et al., 1995) that these architectural assumptions could be made explicit using an Architecture Description Language (or ADL). Architectural descriptions could be used to understand the concepts embodied in component libraries (Perry and Wolf, 1992).

However, there are problems to overcome. ADLs are created for the specification of software architectures, and software architectural styles. They have not been created with component standards like CORBA, ActiveX or Java Beans in mind. The two domains use similar, but incompatible models of components and component bindings, revealed when comparing the Interface Description Languages for components, with the possibilities of the ADL used for software architecture. Moreover, while the mapping of components at the software architectural level to components at the implementation level might be feasible, how other architectural elements should be mapped is unclear.

Furthermore, an architecture describes the system as a whole, while reusable components make use of services provided by the middleware infrastructure and the operating system. In fact, these services should be modeled at the architectural level, to get a real mapping between both levels (Oreizy et al., 1998).

Another question is *how* software architecture and component-based development can be combined. One way of combining these domains is to start with the design of the system's architecture. The architecture should be refined until one is able to choose or build existing components, based upon the architectural specification. These components should be connected according to the architecture. Designing the system's architecture is, in this case, the specification; "filling in" the components is the implementation. However, when one first completes the software architecture without the components at hand in mind, the chance that one can really reuse them is very low. The "inevitable intertwining of specification and implementation" (Swartout and Balzer, 1982) is especially valid when reusable components are involved.

Another way of combining software architecture and component-based development is to build a system using existing components, and describe the architecture of such a system in an Architecture Description Language. The description can be used for analysis.

In this paper, we explore the possibilities of both ways of combining component-based development and software architecture, for the component model of Java Beans. On the one hand, we investigate how we can map a software architecture onto an application of connected Java Beans. We use the framework for classifying ADLs by Medvidovic and Taylor (Medvidovic and Taylor, 1997) to cover the different aspects that might or should be included in an architectural description of a system. On the other hand, we summarize the requirements for an ADL to be able to describe the architecture of an application built by connecting beans.

In section 2, we will give a short overview of the features and concepts of Java Beans. In section 3, we discuss the (im)possibilities of mapping architectural elements onto Java Beans. In section 4, we do the same mapping in reverse. Related work is mentioned in section 5, and in section 6, we discuss how to carry on.

2. JAVA BEANS IN SHORT

Java Beans are pieces of software, written in Java, in such a way that it is possible to build applications by connecting beans, in a "bean-aware" application builder. Such an application builder is able to get information of the bean about its properties, methods, and the events it fires. The user of the application builder may change properties, and connect different beans through events, thus building an application. Everything is done through dragging and dropping, or by filling in property sheets.

In a bean's lifetime, one may discern three different stages. In the first place, a bean should be created. In this paper, we are not concerned with beans programming, and we will just assume the existence of a library of ready-to-use beans. In the second place, a bean is used during the design of an application. The application builder tool discovers its properties, methods and events, the user or developer instantiates the bean, customizes the instances, and connects instances of (the same or different) beans. Of course, a bean may be used multiple times during the design of different applications. The third stage is the existence (as an instance) in a running application. We will refer to the second stage described above as 'design time." A bean in a running application is referred to as "run time."

According to the Java Bean specification (Hamilton, 1997), a Java bean is a reusable software component with at least

- support for introspection. Beans are constructed in such a way that an application builder may discover a bean's properties, methods, and events by introspection.
- support for properties to be inspected or changed: customization. Properties are a bean's appearance and behavior attributes that can be changed at design time.
- support for events, for communication between beans. A bean that wants to receive events (a listener bean) registers its interest with the bean that fires the event (a source bean). Builder tools can examine a bean and determine which events that bean can fire (send) and which it can handle (receive).
- support for persistence. Persistence enables beans to save their state and restore that state later.

A bean interacts with its environment through its set of properties, its set of methods, and the set of events it fires. Properties are attributes that can be read and written. Methods are normal Java methods that can be called from outside the bean. Events that are fired by beans invoke methods in beans that have subscribed on the particular class of events. These beans adhere to the EventListener interface. An event-firing bean and an EventListener bean may be decoupled by placing an EventAdapter bean in between them.

Some properties of event delivery for Java Beans are:

- Event delivery is multicast: one event that is fired invokes an associated method in every bean that has subscribed on the event.
- Event delivery is synchronous with respect to the event source: the associated method in the EventListener bean is executed in the thread of the event-firing bean.
- The set of EventListeners for a certain event may be changed dynamically.

2.1 Example

2.1.1 Connections with events

Imagine a BallThrowing Bean. Throwing a ball is implemented using an event, for which a BallEventObject class is created. When one uses such a Bean in a bean-aware application builder, one may instantiate instances of the BallThrowing bean, and connect them through the BallEvent. A bean (a BallThrowing bean or any other that can handle BallEvents) is connected by stating that the bean listens to BallEvents sent by the BallThrowing bean, and by specifying what action should be taken when receiving a BallEvent.

A BallThrowing bean class should have a list of BallEventListeners, and methods to add and remove objects to and from that list. These methods are used in the application builder, when connections are made and undone.

A BallEventListener bean should have an action method that has a BallEventObject as an argument.

```
public class BallThrower {
   private Vector ballCatchers = new vector();

    public synchronized void addBallEventListener(BallEventListener c) {
      ballCatchers.addElement(c);
    }

   public synchronized void removeBallEventListener(BallEventListener c)
   {
      ballCatchers.removeElement(c);
   }

}

public interface BallEventListener {
   void catchBall(BallEventObject ball);
}

public class BallEventObject extends EventObject {
}
```

2.1.2 Properties

A bean-aware application builder simply searches for set- and get-methods to find the properties of a bean. The BallThrower bean, for instance, could have the number of balls it possesses, as a property:

```
public void setNumber(Integer number) {
  this.number = number;
}

public Integer getNumber() {
  return number;
}
```

Changes to properties may be notified to other beans. Such a property is called **bounded**. A bean with a bounded property maintains a list of PropertyChangeListeners (beans implementing the PropertyCangeListener interface), and it sends a PropertyChangeEvent to those listeneres when the bounded property has been changed.

A property may be **constrained** as well. In this case, the bean maintains a list of VetoableChangeListeners, which are able to check whether a value of the constrained property is within the constraints. The setProperty method of such a bean raises an exception when one of the listeners uses its veto.

A bean-aware application builder recognizes that a property is bounded or constrained, and offers the user of the application builder the possibility to indicate which other beans will act as listeners.

2.1.3 Introspection

In the samples of a bean shown above, we have used conventional names and type signatures of methods and interfaces as a means for introspection. Bean-aware application builders look for set- and get-methods, and addeventlisteners and removeeventlisteners methods, to find the properties of a bean and the events with which it can be connected. A Java Bean may also explicitly specify its properties, events and methods, using a class implementing the BeanInfo class.

2.2 Status and Environment

Because communication between beans consists of event notification and direct method invocation, it is necessary that beans run in the same address space, in this case in the same Java Virtual Machine. Another environmental aspect of beans is that they should assume that they are running in a multithreaded environment: several different threads may simultaneously deliver events, or call methods directly.

Several extensions have been proposed:

– InfoBus (Colan, 1998) from Lotus Development is already available.
  This extension offers a new type of connection between beans: data

flows. Beans may subscribe to certain kinds of data (based on a name). Other beans produce the data. Application builder tools are able to extract from a bean the names of the data it is able to produce. This communication mechanism is known as subscription-based communication (Boasson, 1996). This type of connection is attractive with respect to the introduction of on-line changes (Stuurman and van Katwijk, 1998).

- JavaSpaces (Sun, 1998) is available as a beta version at the moment. JavaSpaces provides a distributed persistence and object exchange mechanism. It is comparable with InfoBus for communication between beans in different Java Virtual Machines.
- An extensible run-time containment and services protocol has been proposed (Cable, 1998). This protocol supports extensible mechanisms that introduce an abstraction for the environment of a bean, enable the dynamic addition of arbitrary services to a bean's environment, provide a mechanism through which a bean may interrogate its environment, and provide a mechanism to propagate an environment to a bean. In short, the notion of the context of a bean is introduced in this extension.
- Another extension is the Java Beans Activation Framework (Calder and Shannon, 1998). This framework supplies the services of determining the type of arbitrary data, encapsulating access to data, discovering the available operations on a particular type of data, and instantiating a software component that corresponds to the desired operation.

3. USING BEANS TO IMPLEMENT AN ARCHITECTURE

The idea of using beans to implement a given software architecture looks promising and desirable: beans are components in the architectural sense of loci of computation and data storage. One has the multi-platform benefits of the Java language; and there is the possibility to have a visual image of the application, consisting of connected components, as a mirror of the software architecture it implements. The idea would be to look for (or build) beans that match the specification of the components of the given architecture, and connect them according to the given configuration.

Which aspects of an architecture are specified depends on the ADL that is used. We will not adhere to one specific ADL, but check the aspects used in the classification framework for ADLs by (Medvidovic and Taylor, 1997). These aspects are: interface, types, semantics, constraints, and evolution of components and connections; composability, heterogeneity, constraints, refinement, scalability, evolution and dynamism of configurations.

When examining the possibility of a mapping between Java Beans and an ADL, we will mainly look at those aspects that are specific to Beans (as opposed to the general aspects of the Java language). Those aspects are the most interesting because they have been specified for the convenience of tool-builders. Next to bean-aware application builders, one may just as easily construct bean-aware software architecture tools.

3.1 Components

The interface of a component in the software architectural sense is the set of interaction points between the component and the external world. An interface specifies the services a component provides, and it might specify the needs of a component. The interface of a Java Bean is its set of properties, its methods, and the events it fires. This information may be extracted from a bean at design time, so one may use an application builder tool to expose the interface. Mapping the interface of a component, specified in an ADL, to the interface of a Java Bean seems rather straightforward, though, of course, not every aspect of an interface that one can specify in an ADL has a counterpart in Java Beans.

ADLs may model abstract components as types, and instantiate them multiple times. Some ADLs allow abstract component types to be parameterized. A Java Bean may be regarded as a parameterized component type insofar as it can be customized. A bean may be instantiated as often as one needs. So, parameterized types are directly supported by Java Beans but not every imaginable component type can be implemented using a Java Bean.

A software architecture specification may contain a model of the component semantics. In a Java Bean, however, semantics are not exposed. When using beans in an application builder, the user is obliged to rely on the documentation supplied with the beans.

An ADL may specify constraints on the abstract state of a component, the implementation, or non-functional properties. With respect to the abstract state of a component, Java Beans have the notion of constrained properties. When such a property is changed another bean validates the change. A mapping between constraints on the abstract state of a component and constrained properties of a bean seems possible.

ADLs may support design evolution through subtyping and refinement. A mapping between such a support and an implementation using Java Beans might be useful for prototyping. However, subtyping and refinement of Java Beans in an application builder is not supported.

In table 1, we summarize which aspects of components, described in an ADL, may be mapped to those aspects of Java Beans that are visible for bean-aware tools.

Table 1: Mapping components in a software architecture description to Java Beans

ASPECT	MAPPING
interface	Possible, beans support a subset
types	Possible, beans support a subset
semantics	Not possible
constraints	Possible, beans support a small subset
evolution	Not possible

3.2 Connections

In an application builder using Java Beans, one glues beans together by connecting them using event notification. An event of an event-firing bean is associated with a method of an event-listening bean. A special case is the notion of constrained properties. A bean with constrained properties is associated with a validator bean. Each time (at run-time) that a property is changed, the change is validated.

InfoBus and JavaSpaces extend this type of connection with the possibility of asynchronous, anonymous data communication. Beans may produce data, and may subscribe to certain kind of data. Producers don't have to wait until every consumer has seen the produced data. Producers and consumers are unaware of each other. Other kinds of connections (create connections for instance) are possible, but cannot be made visible in an application builder, and are "hidden" in the code of the bean.

The interface of a connection in a software architecture is a set of interaction points between the connection and the components attached to it. Each kind of connection that can be used for Java Beans has its own interface: events are of a certain class and should be connected to an eventsource and a set of eventlisteners; InfoBus connections are associated with a name and should be connected to a set of data producers and a set of data consumers. Of course, not every interface that one can specify in an architecture has a counterpart in a Java Beans application.

Some ADLs distinguish connection types from connection instances. Events in Java Beans are always of a certain class, that can be subclassed. So, for event-based connections, one may map the idea of a connection type to an event connection.

Some ADLs provide means to express the semantics of connections. For the connections possible in a Java Beans application, one should specify the semantics of these connections once. Of course, in an architecture, one can specify connections with semantics that have no counterpart in a Java Beans application.

Connection constraints may consist of adherence to interaction protocols, intra-connection dependencies, or usage boundaries. In general, Java Beans give no support to translate these kind of constraints.

Some ADLs provide support for connection evolution, through subtyping or refinement. Again, Java Beans give no support.

Table 2: Mapping connections in a software architecture description to Java Beans

ASPECT	MAPPING
interface	Possible, beans support a subset
types	Possible, beans support a subset
semantics	Not possible (one should first specify beans connections)
constraints	Not possible
evolution	Not possible

3.3 Configurations

With respect to composability, some ADLs support situations where an architecture becomes a component in a bigger architecture. Such a composition can be mirrored in Java Beans, where a composition of interconnected beans may be transformed into one new bean.

Many ADLs offer the possibility to specify global constraints. In general, it will not be possible to map these constraints to visible properties of a Java Beans application.

Darwin, Rapide, and C2 allow specification of dynamism in architectures. Insertion and removal of both components and connections is possible in Java applications, but one cannot extract information about this behaviour by introspection.

Table 3: Mapping configurations in a software architecture description to Java Beans

ASPECT	MAPPING
composition	Possible
constraints	Not possible
evolution	Not possible

4. USING AN ADL TO DESCRIBE A BEANS CONFIGURATION

The previous section showed that not every software architecture can be mapped onto a configuration of Java Beans. Not every part of an architecture description is translatable into either a Java Bean or a connection between beans. When using beans to construct a system based on a certain software architecture, one should check the types of components and the types of connections.

Automating such a process is only attractive when one conforms to the subset of architectures that can be implemented using beans. On the other hand, it seems to be the case that an application built by connecting Java Beans may be translated relatively easily into an architectural description. One should choose an ADL based on how much of the information, available in a beans application, can be described. In the remainder of the section, we make use of the classification of (Medvidovic and Taylor, 1997), for ADLs, and we take only those ADLs into account that are part of the survey : Aesop, MetaH, LILEANNA, ArTek, C2, Rapide, Wright, UniCon, Darwin, SADL and ACME.

4.1 Beans

The properties, methods and the events a bean can fire, should be translated into an interface specification. All ADLs support specification of component interfaces.

The language should provide the means to specify parameterized types, with the properties that can be changed at design time as parameters. Only ACME, Darwin and Rapide make explicit use of parameterization.

Bounded properties may be translated into constraints on the abstract state of a component. Rapide uses an algebraic language to specify constraints on the abstract state of a component.

Table 4: Mapping aspects of beans to an ADL

ADL	INTERFACE SPECIFICATION	PROPERTIES AS PARAMETER	BOUNDED PROPERTIES
Aesop	yes	no	no
MetaH	yes	no	no
LILEANNA	yes	no	no
ArTek	yes	no	no
C2	yes	no	no
Rapide	yes	yes	yes
Wright	yes	no	no
UniCon	yes	no	no
Darwin	yes	yes	no
SADL	yes	no	no
ACME	yes	yes	no

4.2 Connections

Connections between an event source and an event listener should be translated into a specification of a connection with the appropriate interface. The same applies for the dataflow connections of the InfoBus and JavaSpaces extension. This is possible in all of the surveyed ADLs.

The semantics for the Java Beans-style event-based and dataflow connections should be expressed in the ADL. It should be possible to express other kind of connections too, when future extensions introduce new types of connections. Rapide, Wright, and UniCon support such specifications.

Table 5: Mapping aspects of connections of Beans to an ADL

ADL	EVENTS	DATAFLOW	SEMANTICS
Aesop	yes	yes	no
MetaH	yes	yes	no
LILEANNA	yes	yes	no
ArTek	yes	yes	no
C2	yes	yes	no
Rapide	yes	yes	yes
Wright	yes	yes	yes
UniCon	yes	yes	yes
Darwin	yes	yes	no
SADL	yes	yes	no
ACME	yes	yes	no

4.3 Configurations

Since it is possible to compose beans into one bigger bean, an ADL used to describe a bean-based application should support such kind of composition. Most ADLs do support it.

Because Java Beans is still developing, and more extensions are to be expected, an ADL should allow for such extensions.

Table 6: Mapping aspects of configurations of Beans to an ADL

ADL	COMPOSITION
Aesop	no
MetaH	yes
LILEANNA	no
ArTek	no
C2	yes
Rapide	yes
Wright	yes
UniCon	yes
Darwin	yes
SADL	yes
ACME	yes

4.4 Implicit Aspects

Above, we described the possibilities of different ADLs to describe those aspects of Java Bean-based applications that are visible for "bean-aware" tools. However, some implicit aspects of Java Beans should be described too, when distilling the architecture of an application. To name a few:

- Threads. Every Java Bean may run in its own thread. At the same time, its methods may be called by other beans, and executed in the thread of the caller. A software architecture description of a beans application should specify this aspect, though it is not available through introspection.
- Create-connections. A bean may instantiate other beans at run-time. Such a connection should certainly be described, but again, information about these relationships is not available through introspection.
- Run-time change of the configuration. Apart from the possibility to create new instances of beans at run-time, beans are also able to change the connections at run-time. This will especially be seen very often in applications based on the Activation Framework extension.

At this moment, information about these possibilities is not available for application builder tools. However, because the run-time flexibility of the

Java system is one of its advantages, an ADL for the description of beans applications should preferably support the specification of dynamism in the configuration. These ADLs are Darwin, Rapide and C2.

Table 7: Mapping implicit aspects of Beans to an ADL

ADL	RUN-TIME CHANGE
Aesop	no
MetaH	no
LILEANNA	no
ArTek	no
C2	yes
Rapide	yes
Wright	no
UniCon	no
Darwin	yes
SADL	no
ACME	no

5. RELATED WORK

Reuse of Off-The-Shelf components in combination with the C2 style has been explored in (Medvidovic et al., 1997). They constructed a Class Framework of reusable classes that can be used to implement C2 style architectures, and integrated several OTS components with the C2 style. This integration was done by wrapping OTS objects in C2 components, and mapping events into C2 messages and vice-versa. In this work, the C2 style is the point of departure, and reusable components are adapted in such a way that they can be used to implement C2 style architectures.

A tool to detect architectural mismatches during design has been constructed by Abd-Allah (Abd-Allah, 1996). His method is based on the notion of "conceptual features", which can be used to detect architectural mismatches. The goal of this work is to enhance the possibilities of reusing components, by scanning them on assumptions with respect to these features.

6. DISCUSSION

In this paper, we have made a start on combining Java Beans and software architecture.

6.1 Combining Software Architecture and Beans

As we have seen, it is highly improbable that a certain software architecture can be mapped to an application built by connecting existing beans unless the designer of the architecture has taken such an implementation into account. A more feasible approach to combine beans and software architecture is to build a system using beans, and describe the system's architecture using an ADL. In that case, by choosing beans as components, one restricts oneself to a certain subset of architectural elements.

However, as we have seen, not all the necessary information to describe an architecture can be extracted from beans and their connections. Certain aspects are implicit, and can only been revealed by inspecting the code of the beans in use. Automating such a process is only feasible when beans adhere to standard conventions for the implementation of these aspects. In fact, this would be an extension to the Java Beans specification.

One can imagine an intermediate approach: using beans, especially developed for this purpose, to construct the system's architecture, and implementing the system using beans that are specialized versions of the "design" beans. Such an approach would benefit of an extension where one can classify beans as being a specialization of another bean.

Neither of these approaches comes for free: we have to extend the standard for Java Beans to achieve a tight relationship between the software architecture description of a system and its implementation using beans. On the other hand, the Java Beans specification already offers substantial support for extracting an architectural description: the property of introspection, meant for application builder tools, can be used for a translation into an ADL of the exposed features of a bean.

6.2 Design for Change

An attractive property of both approaches is that changes in the software are automatically handled at the architectural level. On-line change capabilities are needed in several domains (see for instance (Stankovic, 1996)), and the ideal situation would be that such changes can be applied at the architectural level.

Prerequisites for a system with on-line change capacities at the architectural level are:

- The software architecture is reflected in the executable. Parts of the executable from which components can be instantiated are traceable and replaceable.
- Components may be added, deleted or replaced, at execution time.

- Bindings of components through connections occur dynamically. In other words, connections may be added, deleted or replaced at execution time.
- Instantiation of components and connections is possible from outside the system.
- The functionality of components is not directly dependent on other components.
- It is possible to analyze properties of the system at the architectural level. Before a change is applied, the architecture should be analyzed to guarantee that the changed system will meet the changed requirements.

Obviously, using a method based on the combination of a software architecture description and a Java Beans application, it is relatively easy to build systems with on-line change capacities on the architectural level.

REFERENCES

Abd-Allah, A. (1996) Composing Heterogeneous Software Architectures, PhD Dissertation, Center for Software Engineering, University of Southern California. *http://sunset.usc.edu/~aabdalla/aaadef.ps*.

Allen, R. and Garlan, D. (1994) Beyond Definition/Use: Architectural Interconnection, in *Proceedings of the Workshop on Interface Definition Languages*, Portland, Oregon, January.

Boasson, M. (1996) Subscription as a Model for the Architecture of Embedded Systems, in *Proceedings of the 2nd IEEE Conference on Engineering of Complex Computer Systems*, Montreal, Canada.

Cable, L. (1998) A Draft Proposal to define an Extensible Runtime Containment and Services Protocol for JavaBeans (Version 0.98). Sun Microsystems.

Calder, B. and Shannon, B. (1998) JavaBeans Activation Framework Specification (Version 1.0). Sun Microsystems.

Colan, M. (1998) InfoBus 1.1 Specification. Sun Microsystems.

Garlan, D. and Allen, R. and Ockerbloom, J. (1995) Architectural Mismatch or Why it's hard to build systems out of existing parts, in *Proceedings of the International Conference on Software Engineering*, Seattle, April.

Garlan, D. and Shaw, M. (1993) An Introduction to Software Architecture, in *Advances in Software Engineering and Knowledge Engineering, volume 1* (ed. V. Ambriola and G. Tortora), World Scientific Publishing Company, New Yersey.

Hamilton, G. (Editor) (1997) JavaBeans 1.01 API Specification. Sun Microsystems.

Medvidovic, M. and Oreizy, P. and Taylor, R.N. (1997) Reuse of Off-The-Shelf Components in C2-Style Architectures, in *Proceedings of the 1997 Symposium on Software Reusability,* Boston, pp 190-198.

Medvidovic, M. and Taylor, R.N. (1997) A Framework for Classifying and Comparing Architecture Secription Languages, in *Proceedings of the 6th European Software Engineering Conference, Lecture Notes in Computer Science*, **1301**, 60-76.

Oreizy, P. and Medvidovic, N. and Taylor, R.N. and Rosenblum, D.S. (1998) Software Architecture and Component Technologies: Bridging the Gap, in *Proceedings of the OMG-DARPA Workshop on Compositional Software Architectures*, Montery, CA, January 6-8.

Perry, D.E. and Wolf, A.L. (1992) Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, **vol 17, nr 4**, 40-52.

Shaw, M. and DeLine, R. and Klein, D.V. and Ross, Th.L. and Young, D.M. and Zelesnik, G. (1995) Abstractions for Software Architecture and Tools to Support them. *IEEE Transactions on Software Engineering*, **April**, 314-335.

Stankovic, J.A. (1996) Real-time and Embedded Systems. Group Report of the Real-Time Working Group of the IEEE Technical Committee on Real-Time Systems. *http://www-ccs.cs.umass.edu/sdcr/rt.ps*

Stuurman, S. and van Katwijk, J. (1998) On-line Change Mechanisms, the Software Architectural Level, to appear in *Proceedings of the the 6th International Symposium on the Foundations of Software Engineering,* Orlando.

Sun Microsystems Inc. (1998) JavaSpaces Specification, Revision 1.0 Beta. *http://www.javasoft.com/products/jini/specs/javaspaces.pdf*

Swartout, W. and Balzer, R. (1982) On the Inevitable Intertwining of Specification and Implementation. *Communications of the ACM*, **vol 25, nr 7**, 438-440.

ARCHITECTURAL PATTERNS AND STYLES

Architectural Styles as Adaptors[1]

Don Batory [i], Yannis Smaragdakis [i] & Lou Coglianese [ii]

Department of Computer Sciences, The University of Texas, Austin, TX 78712 [i] *&*
LGA, Inc.,12500 Fair Lakes Circle, Suite 130, Fairfax, Virginia [ii]
{batory, smaragd}@cs.utexas.edu, lou@lga-inc.com

Keywords: architectural styles, product-line architectures, GenVoca model, adaptors, software refinements, program transformations.

Abstract: The essence of architectural styles is component communication. In this paper, we relate architectural styles to adaptors in the GenVoca model of software construction. GenVoca components are refinements that have a wide range of implementations, from binaries to rule-sets of program transformation systems. We explain that architectural styles can (1) be understood as refinements (like other GenVoca components) and (2) that they are generalizations of the OO concept of adaptors. By implementing adaptors as program transformations, complex architectural styles can be realized in GenVoca that simply could not be expressed using less powerful implementation techniques (e.g., object adaptors). Examples from avionics are given.

1 INTRODUCTION

McIlroy and Parnas observed almost thirty years ago that software products are rarely created in isolation; over time a family of related products eventually emerges (McIlroy, 1968 and Parnas, 1976). Software design and development techniques then were aimed at one-of-a-kind products. While software design methodologies have improved significantly both in quality and sophistication, one-of-a-kind products are still the norm. However, it is

1. This paper is derived from two ADAGE technical reports (Batory and Coglianese, 1993; Batory and Smaragdakis, 1995) that were sponsored by U.S. D.o.D. A.R.P.A. in cooperation with the U.S. Wright Laboratory Avionics Directorate under contract F33615-91C-1788. This research is sponsored in part by Microsoft, Schlumberger, the University of Texas Applied Research Labs, and U.S. D.o.D. A.R.P.A. under contract F30602-96-2-0226.

becoming increasingly apparent that product families are indeed very common and methodologies are needed to accommodate their economical design and construction.

A *product-line architecture (PLA)* is a blue print for building a family of related applications. A number of different approaches for designing PLAs have been under development for some time, each proffering many successes (Weiss, 1990; Cohen et al., 1995; Harrison and Ossher, 1993; Batory and O'Malley, 1992). Of these approaches, the *GenVoca* approach is distinguished by components that export and import standardized interfaces (Batory and O'Malley, 1992; Smaragdakis and Batory, 1998). Applications of a product-line are assembled purely through component composition. Components themselves can encapsulate domain-specific "intelligence" that can, for example, automate domain-specific optimizations that are critical to application performance.

A fundamental issue in composing applications from components has to do with the way components communicate their needs and results. This is what we consider the essence of *architectural styles*: the separation of a component's computations from the means by which it communicates. As no single architectural style suffices for all applications, there needs to be a way in which styles can evolve (or be replaced) within or across application instances.

In this paper, we explore the relationship of architectural styles and GenVoca. GenVoca components are refinements that have a wide range of implementations, from binaries to rule-sets of program transformation systems. Architectural styles can also be understood as refinements and treated just like other GenVoca components. Furthermore, style refinements are actually generalizations of the OO concept of adaptors. By implementing adaptors as program transformations, complex architectural styles can be realized in GenVoca that simply could not be expressed using less powerful implementation techniques (e.g., object adaptors). Examples from avionics are given to partially support this claim.

2 COMPONENTS, ARCHITECTURAL STYLES, AND REFINEMENTS

The term *software architecture* refers to an abstract model of an application that is expressed in terms of intercommunicating components. Components communicate via abstract conduits whose implementations are initially unspecified. An *architectural style* is an implementation of a conduit; the original vision of Garlan and Shaw allowed software architects to select different styles (conduit implementations), such as pipes, RPC, etc., that would satisfy application performance or functionality constraints. Some architectural styles could reveal lower-level components and conduits, thereby allowing conduit implementations to be expressed in a progressive or "layered" manner (Gorlick and Razouk, 1991).

Mapping an abstract concept or declaration to a concrete (or less abstract) realization is a *refinement*. Just as architectural styles are refinements of communication conduits, component implementations can also be revealed as a progression of refinements. Such progressions are sometimes, but not always, equivalent to "layered" implementations.

Refinements expose the implementations of components and conduits in a uniform way (which seems reasonable, since both are expressed as software). It is evident that a powerful model of software architectures can be created around the concept of refinements as primitive building blocks of applications. This is one of several basic ideas that underly the GenVoca model of software construction.

3 A MODEL OF PRODUCT-LINE ARCHITECTURES

A premise of GenVoca is that plug-compatible and interchangeable software "building blocks" can be created by standardizing both the fundamental abstractions of a mature software domain *and* their implementations. The number of abstractions in a domain is typically small, whereas a huge number of potential implementations exist for every abstraction. GenVoca advocates a layered decomposition of implementations, where each layer or component encapsulates the implementation of a primitive feature shared by many applications. The advantage is *scalability* (Batory, et al., 1993; Biggerstaff, 1994): libraries have few components, while the number of possible *combinations* of components (i.e., distinct applications in the domain that can be defined) is exponential. GenVoca has been used to create product line architectures for diverse domains including avionics, file systems, compilers, and network protocols (Coglianese and Szymanski, 1993; Heidemann and Popek, 1993; Hutchinson and Peterson, 1991).

Components and realms. A hierarchical application is defined by a series of progressively more abstract virtual machines (Dijkstra, 1968). (A *virtual machine* is a set of classes, their objects, and methods that work cooperatively to implement some functionality. Clients of a virtual machine do not know how this functionality is implemented). A *component* or *layer* is an implementation of a virtual machine. The set of all components that implement the same virtual machine is a *realm*; effectively, a realm is a library of interchangeable components. In Figure 1a, realms S and T have three components, whereas realm W has four.

```
(a)     S={a,b,c}                  (b)     S:= a   | b   | c   ;

        T={d[S],e[S],f[S]}                 T:= d S | e S | f S ;

        W={n[W],m[W],p,q[T,S]}             W:= n W| m W  | p | q T S;
```

Figure 1: Realms, components, and grammars

Parameters and refinements. A component has a (realm) parameter for every realm interface that it imports. All components of realm T, for example, have a single parameter of realm S.[2] This means that every component of T exports the virtual machine of T (because it belongs to realm T) and imports the virtual machine interface of S (because it has a parameter of realm S). Each T component encapsulates a *refinement* between the virtual machines T and S. Such refinements can be simple or they can involve domain-specific optimizations and the automated selection of algorithms.

Applications and type equations. An *application* is a named composition of components called a *type equation*. Consider the following two equations:

```
A1 = d[ b ];
A2 = f[ a ];
```

Application A1 composes component d with b; A2 composes f with a. Both applications are equations of type T (because the outermost components of both are of type T). This means that A1 and A2 implement the same virtual machine and are interchangeable implementations of T. Note that composing components is equivalent to stacking layers. For this reason, we use the terms component and layer interchangeably.

Grammars, product lines, and scalability. Realms and their components define a grammar whose sentences are applications. Figure 1a enumerated realms S, T, and W; the corresponding grammar is shown in *Figure 1b*. Just as the set of all sentences defines a language, the set of all compo-

2. Components may have many other parameters in addition to realm parameters. In this paper, we focus only on realm parameters.

nent compositions defines a *product-line*. Adding a new component to a realm is equivalent to adding a new rule to a grammar; the family of products that can be created enlarges automatically. Because huge families of products can be built using few components, GenVoca is a *scalable* model of software construction.

Symmetry. Just as recursion is fundamental to grammars, recursion in the form of symmetric components is fundamental to GenVoca. More specifically, a component is *symmetric* if it exports the same interface that it imports (i.e., a symmetric component of realm `W` has at least one parameter of type `W`). Symmetric components have the unusual property that they can be composed in arbitrary ways. In realm `W` of Figure 1, components `n` and `m` are symmetric whereas `p` and `q` are not. This means that compositions `n[m[p]]`, `m[n[p]]`, `n[n[p]]`, and `m[m[p]]` are possible, the latter two showing that a component can be composed with itself. Symmetric components allow applications to have an open-ended set of features (because an arbitrary number of symmetric components can appear in a type equation).[3]

Design rules, domain models, and generators. In principle, any component of realm `S` can instantiate the parameter of any component of realm `T`. Although the resulting equations would be *type correct,* the equation may not be *semantically* correct. That is, there are often domain-specific constraints that instantiating components must satisfy *in addition to* implementing a particular virtual machine. These additional constraints are called *design rules. Design rule checking (DRC)* is the process of applying design rules to validate type equations (Batory and Geraci, 1997). A GenVoca *domain model* or *product-line architecture (PLA)* consists of realms of components and design rules that govern component composition. A *generator* is an implementation of a domain model; it is a tool that translates a type equation into an executable application.

Implementations. A GenVoca model is an abstract description of a product-line architecture. It expresses the primitive building blocks of a PLA as composable refinements (components). The model itself does not specify *when* refinements are composed or *how* they are to be implemented. Refinements may be composed *statically* at application-compile time or *dynamically* at application run-time. Refinements themselves may be implemented *compositionally* (e.g., COM binaries, Java packages, C++

3. We refer to virtual machines as "standardized interfaces". However, these interfaces are not immutable; they can change with the addition or removal of a component (Batory and Geraci, 1997; Smaragdakis and Batory, 1998). Thus, symmetric components can add new functionalities that are reflected in application interfaces.

templates), as *metaprograms* (i.e., programs that generate other programs by composing prewritten code fragments), or as rule-sets of *program transformation systems (PTSs)*. Compositional implementations offer no possibilities of static optimizations; metaprogramming implementations automate a wide range of common and simple domain-specific optimizations at application synthesis time; PTSs offer unlimited optimization possibilities. Choosing between dynamic and static compositions, and alternative implementation strategies is largely determined by the performance and behavior that is desired for synthesized applications.

Separating PLA design from implementation provides a significant conceptual economy: GenVoca offers a *single* way in which to conceptualize building-block PLAs and *many* ways in which to *realize* this model (each with known trade-offs).

4 ARCHITECTURAL STYLES AS ADAPTORS

4.1 Motivation

An *architectural style* refers to the means by which components communicate their needs and results, as well as a set of constraints that govern the overall constellation of an application's components. For example, components can communicate through pipes in the pipe-and-filter style; constellations are largely limited to linear chains of components. Our focus on architectural styles lies exclusively with component communication. Note that this definition of a "style" is not as broad as that in the treatment of architectures by Perry and Wolf (Perry and Wolf, 1992) (where a style can be any abstract architectural element and may cover as many aspects as an entire architecture), but follows a more constrained view taken by other researchers (Shaw and Clements, 1997; DeLine, 1996).

The obvious first question is, why use different architectural styles? There are many reasons, some of which are:

- *Compatibility reasons*. Most often, a style is fixed by convention or because the need to distinguish between computation and communication had not become apparent at component implementation time. Thus, components need to adopt a special style to communicate with existing software. The scale of both components and interfaces may vary widely. Many standard protocols (interprocess communication, windowing application conventions, COM for ActiveX controls) can be viewed as alternative styles for connectors to some unit of functionality.

- *Performance/portability reasons*. Even simple decisions at the implementation level can constitute stylistic dependencies: a piece of code could be inlined or made into a procedure. A set of parameters may be passed through global variables or procedure arguments. A service can be implemented as a static or dynamic library, or even a stand-alone server. Such decisions fundamentally affect the performance and portability of a component. Distributed applications are a good example. Deciding whether a piece of functionality is local or accessed over a network can be viewed as a simple stylistic choice, albeit one that fundamentally affects performance. Ideally the same component could adopt different styles and be used in vastly different applications. For instance, the same piece of functionality may be in the core of both an embedded system (with a primitive OS, small memory, and slow processor) and a high-end server system. The component should not have to be rewritten but should automatically adapt (through a style adaptor) to the capabilities of either runtime environment.

4.2 GenVoca and adaptors

GenVoca components are designed *a priori* to communicate with their clients in one style. For example, application `A1` of Section 3 has component `d` communicating with component `b` via the `S` interface. What exactly the mechanisms and protocols are (e.g., local procedure calls, marshalled arguments, global-variables, etc.) is governed by the definition of the virtual machine `S`. But suppose we would like component `d` to communicate with `b` via another style — remote procedure calls — which we would encode as some interface `G`. Furthermore, we would like components `d` and `b` to remain unchanged, so that `d`'s calls to interface `S` are translated (refined) into calls to interface `G`; similarly, invocations of `G` methods are translated (refined) into invocations of `S` methods for `b` to process, and vice versa.

This can be accomplished using *adaptors* (Gamma, et al., 1994). For our example, we need to add two components and one realm to *Figure 1*. Component `s2g[G]` would translate (refine, adapt) `S` method invocations to `G` method invocations; `s2g[G]` would be a new member of realm `S`. Component `g2s[S]` would do the opposite: it would translate (refine, adapt) calls to `G` into calls to `S`; `g2s[S]` would be the (lone) component of a newly-created realm `G`. *Figure 2* graphically illustrates the modification of `A1` to `A1'`, where `d` indirectly communicates (via interface `G`) with `b`.

Note the following. First, the essence of replacing one architectural style with another should not alter the semantics of the target application. We have indeed not altered the computations of `A1` in any way by rewriting it as

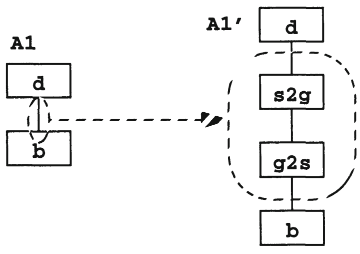


Figure 2: Changing architectural styles

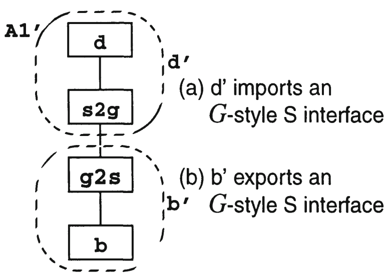

A1'
d
s2g
d'
(a) d' imports an G-style S interface
g2s
b
b'
(b) b' exports an G-style S interface

Figure 3: Stylized component interfaces

`A1'`; the only thing that has changed is the means by which components `d` and `b` communicate. The *architectural style equation* `G-Style[x] = s2g[g2s[x]]` is the identity mapping, and algebraically `A1` = `A1'`. In general, we postulate that architectural styles are algebraic identity elements. Given the type equation of an application, it is possible to rewrite the equation in many different ways using 'style' identities. Each equation would describe a different implementation of that application — i.e., the same fundamental computations are performed in the same order, the only difference is the means by which components communicate.

Second, one of the goals of component-based design is to avoid component replication in library development. Replication occurs, for example, when the computations of a component are fused with its communication style. Different encodings of a computation exist when multiple styles need to be supported. Unfortunately, this approach doesn't scale. If there are *n* computations and *s* styles, then potentially *n*s* different components may be needed. Adding a new style may introduce *n* components; adding a new computation might introduce *s* components.

Our model suggests a way to avoid such replication. Components and adaptors are designed to be orthogonal to each other; this gives them a mix-and-match quality that avoids the fusing of component computations with communication styles. In *Figure 3*, we can view application `A'` as a composition of components `d'` and `b'`, where `d'` communicates with `b'` via interface `G` (i.e., the computations of `d` and `b` are communicating via a "`G`" style). Algebraically, `d'[x] = d[s2g[x]]` and `b' = g2s[b]`.

This view of architectural styles as adaptors is not novel. Nevertheless, standard compositional implementations of adaptors (e.g., as objects, procedures, or templates) have not always been up to the task. The use of adaptors makes interface translations look conceptually trivial but the implementations of such translations may be very sophisticated. Compositional implementations (e.g., OO object adaptors) are inadequate to equate architectural styles with adaptors. There are many architectural styles that cannot be implemented (or implemented efficiently) in this manner. (Consider the example given earlier, of a single component being used in both a high-end server and an embedded system.) This is not surprising: *the use of a compositional mechanism (e.g., procedures or objects) is itself a stylistic dependency*!

In contrast, our approach focuses on conceptualizing building-blocks of product-line architectures as refinements. The advantage of refinements is that they are not limited to compositional implementations. In fact, many of the useful expressions of styles as adaptors employ metaprogramming tools (software generators). Generators have control over components that exceeds the limits of languages. For instance, code fragments can be fused together (Smaragdakis and Batory, 1997) or specialization hooks can be eliminated from the generated code if they are not used. Even very simple "generators" (like the Microsoft MFC and ATL wizards for adapting software to the style of Windows applications, ActiveX controls, etc.) are much more powerful than a simple collection of compositional components. It is this flexibility of generators that allows us to equate architectural styles with ("intelligent") adaptors.

A significant consequence of using software generators is that the structure of the generated program may look nothing like the structure of its specification. Hence, even though GenVoca is a layered model, it is not constrained to building layered implementations. GenVoca just offers the "vocabulary" for specifying product-lines. Generators are compilers that translate such specifications into their concrete realizations. A layered specification may well be describing programs with non-layered architectural styles (e.g., client-server, blackboard, etc.).

5 AN EXAMPLE FROM AVIONICS

ADAGE was a project to realize a GenVoca-based product-line architecture for avionics (in particular, navigation) software (Coglianese and Szymanski, 1993; Batory and Smaragdakis, 1995). While the details of the model are not germane to this paper, the central idea is that navigation components communicate by exchanging state vectors — i.e., run-time objects that encode information about the position of an aircraft at a particular point in time. Different components perform common computations on state vectors (e.g., filtering, integration, etc.). This section overviews an approach that was prototyped for ADAGE.

For the purposes of our paper, we will study a very simple type equation, `E = Main[A[B[C]]]`, that is a linear chain of components. The `Main` component encapsulates the application that is periodically executed; the remaining components perform computations on state vectors. Computations proceed bottom-up; that is, component `C` outputs a vector that is processed by `B`; `B`'s vector is processed by `A`; `Main` displays the contents of `A`'s vector. The specific computations will be abstracted into a set of uninterpreted algorithms that will allow us to explore the impact of using different architectural styles. Each component exports a read-vector method that a higher-level component can call. Although there are many other methods, the central idea of architectural styles can be conveyed with the rewriting/packaging of this one method; other methods can be treated in a similar manner. Note that our examples are deliberately idealized with complicating details omitted.

We will denote the read-vector computation of component `C` to be algorithm `c()`; that is, whenever the read-vector computation of `C` is called (no matter how the read-vector method is expressed), algorithm `c()` is invoked. Similarly, the read-vector computation of component `B` is algorithm `b(x:TYPE_C)`, where `TYPE_C` is the type of vector output by component `C`. The read-vector computation of `A` is algorithm `a(x:TYPE_B)`, where `TYPE_B` is the type of vector output by component `B`.

5.1 Example styles

There are many ways of encoding the computations of `E` as one or more Ada tasks. Many reflect minor differences in programming styles. In this section, we present three very different implementations of `E` — **executive**, **layered**, and **task** — each with its own unique architectural style. Every implementation executes *exactly* the same domain-specific computations in the same order; the only difference is how the components of `E` communicate with each other (and hence are encoded). Later, we will explain how each of these implementations could be "derived" or "generated" using GenVoca architectural-style adaptors.

Executive implementation. The most common way in which the computations of `E` are realized in avionics software is as an *executive* (also commonly known as *time-line executive*). The state vector that is output by each component is stored in a global variable; read-vector methods are encoded as procedures that read and write global state vectors. The `Main` task executes read-vector methods in an order that reflects a bottom-up evaluation of `E`. An Ada representation of an **executive** encoding of `E` is depicted in *Figure 4*.

```
-- global state vectors

vec_a : TYPE_A;
vec_b : TYPE_B;
vec_c : TYPE_C;

-- read-vector for component C

procedure READ_C is
begin
   vec_c = c();
end;

-- read-vector for component B

procedure READ_B is
begin
   invec : TYPE_A;
   invec = vec_c;
   vec_b = b( invec );
end;

-- read-vector for component A

procedure READ_A is
begin
   invec : TYPE_B;
   invec = vec_b;
   vec_a = a(invec);
end;

-- main task

task body MAIN is
begin
   x : integer;
   loop
-- bottom-up evaluation of E
      READ_C;
      READ_B;
      READ_A;
-- compute time x till next cycle
      delay x;
   end loop
end;
```

Figure 4: The "Executive" Style

```
-- component read functions

function READ_C return TYPE_C is
begin
   return c();
end;

function READ_B return TYPE_B is
begin
   invec : TYPE_B;

   invec = READ_C;
   return b(invec);
end;

function READ_A return TYPE_A is
begin
   invec : TYPE_B;

   invec = READ_B;
   return a(invec);
end;

-- main task

task body MAIN is
begin
   x : integer;
   vec_a : TYPE_A;
   loop
      vec_a = READ_A;
      -- compute time x till next cycle
      delay x;
   end loop
end;
```

Figure 5: The "Layered" Style

Layered implementation. A typical layered implementation of **Main** would permit **Main** to call only the methods of component **A**; **A**'s methods, in turn, would call methods of component **B**, and **B**'s methods would call those of **C**. State vectors are returned as method results; there are no global variables. An Ada representation of a **layered** encoding of **E** is depicted in *Figure 5*.

Task implementation. A third and very different implementation of **E** would be to realize each component as an Ada task; state vectors would be exchanged between tasks. *Figure 6* depicts a **task** encoding of **E**.

```
-- components as tasks

task TASK_C is
  entry READ_C(vec_c : out TYPE_C);
  ...
end;
task body TASK_C is
begin
  loop
    accept READ_C(vec_c : out TYPE_C) do
      vec_c = c();
    end;
    ...
  end loop
end


task TASK_B is
  entry READ_B(vec_b : out TYPE_B);
  ...
end;
task body TASK_B
use TASK_C is
begin
  loop
    accept READ_B(vec_b : out TYPE_B) do
      invec : TYPE_C;

      -- read vector from TASK_C
      TASK_C.READ_C(invec)
      vec_b = b(invec);
    end;
    ...
  end loop
end;
```

```
task TASK_A is
  entry READ_A( vec_a : out TYPE_A );
  ...
end;
task body TASK_A
begin
  loop
    accept READ_A(vec_a:out TYPE_A)
    do
      invec : TYPE_B;

      -- read vector from TASK_B
      TASK_B.READ_B(invec);
      vec_a = a(invec);
    end;
    ...
  end loop
end


-- main task

task body MAIN
use TASK_A is
begin
  x : integer;
  invec : TYPE_A;
  loop

    -- read vector from TASK_A
    TASK_A.READ_A(invec);
 -- compute time x till next cycle;
    delay x;
  end loop
end;
```

Figure 6: A transducer/task style

Note that all three of the above examples are semantically equivalent (i.e., they each perform exactly the same computations in the same order), *and are syntactic transformations of each other.* The only code that is shared among all three are the algorithms **`c()`**, **`b(x:TYPE_C)`**, and **`a(x:TYPE_B)`**; the differences are simply in the packaging of these algorithms in a particular architectural style.

There are several trade-offs involved in choosing one of the above styles. Not all of them are apparent in our presentation of these styles as Ada code fragments. Nevertheless, we will try to outline here the trade-offs between the "executive" and "task" implementations.

Time-line executive is the easiest runtime implementation to write. The programmer needs to set a timer interrupt for the basic system cycle. When the timer goes off, a predefined set of procedures that implement the application functions get called. The main advantage of this style is its predictability. Application functions will run in a fixed pattern. Adding the maximum time for each function yields the maximum time for the cycle.

The simplicity of the dispatcher (no scheduler is needed) results in a low overhead, quite predictable OS when no real-time alternative exists. The down side to the executive style is that it is too simplistic. The data used by the system is fundamentally produced at different rates. Computations need to run at a variety of rates. Data consumers need information with another set of rates and latencies. If some unit needs to operate at a rate different than the basic cycle, the system will become more complex. Adding and deleting functions or changing the timing requirements forces one to modify code throughout the system. In all, the code is partitioned more to satisfy timing than based on objects or functional cohesion. A second problem arises from the linear nature of the executive's calling sequence. Data is not passed from one part of the cycle to the next. Rather the majority of state information is stored in global data. Without formal data-flow analysis, it is easy to use data in global variables that have not yet been updated for the current cycle.

Tasking architectures have been designed to overcome the brittle, error-prone nature of time-line executives. Modern schedulers permit analysis to prove that all processing deadlines will be met. Thus data can be produced at the required rates. Tasks can be added and the effects of their load on the system can be calculated. The disadvantage of the task style is that it is difficult to implement and generally has a higher overhead.

In the next section, we explain how computations and "style" adaptors can be packaged as GenVoca components.

5.2 Packaging adaptors as components

As mentioned earlier, both components and adaptors that represent architectural styles can be unified by the concept of consistent refinements. An implementation of refinements that can synthesize the examples of Section 5.1 are metaprograms and rule-sets of program transformation systems (PTS). A *metaprogram* is a program that generates another program by composing code fragments; a rule-set of a PTS is a set of tree rewrite rules that, when applied, progressively transform one program into another. For both metaprograms and PTS, programs are manipulated as data. We will explain our implementation using a metaprogramming approach. Later in Section 5.3.2, we motivate the generalization to rule-sets of PTSs.

Our model assumes that components communicate in a predetermined "standard" style. Any other style would be obtained through the use of adaptors. For this to be possible, each avionics component will be represented as a metaprogramming protocol — each component can query the capabilities and properties of adjacent components to determine what code should be generated. In particular, this allows each component to determine (a) the global variables that are to be used, (b) the protocol on how a component's current state vector is to be obtained, (c) when component methods are to be executed, and (d) what interface "wrapper" should surround the source code of domain-specific computations. Each of these capabilities will be expressed as methods that return code fragments.

5.2.1 An executive component

Let's look at how component `A` might be represented as a metaprogram. Let's assume that the "standard" style in our model is **executive** (any style will do). So our implementation of component `A` will encapsulate *both* A's fundamental computations as well as its **executive** encoding. The following explains a set of methods that `A` (as well as `B` and `C`) would implement:

- **global-variable method**: This method outputs the declaration of any global variable of a component. Component `A` would output "`A_vec : TYPE_A;`". That is, it would output a standard name for its global variable (`A_vec`) and its declaration. In addition, the global-variable method of the component beneath `A` would be invoked, thereby generating a chain of global variable declarations originating from multiple components. Consider equation `E`. When the global-variable method for `A` is called, the following declarations would be generated:

```
vec_a : TYPE_A;
vec_b : TYPE_B;
vec_c : TYPE_C;
```

- **get-current-vector method**: This method outputs a statement that assigns local variable `invec` to the current vector of the given component. For component `A`, the statement "`invec = vec_a;`" is produced, meaning that the current vector of `A` is in global variable `vec_a`.

- **interface-generation method**: This method generates a component's read-vector method in executive style. Component `A` produces a parameterless procedure where the body of the procedure invokes algorithm `a(x:TYPE_B)`:

```
procedure READ_A is
begin
   invec : TYPE_B;
   --- set invec to appropriate value
   vec_a = a(invec);
end
```

Note that the above procedure is incomplete, because **`invec`** has yet to be initialized. The assignment statement that initializes **`invec`** is produced by invoking the **`get-current-vector`** method of the component that lies immediately beneath **`A`**. Again consider equation **`E`**. The read procedure that is generated by calling **interface-generation** for component **`A`** is:

```
procedure READ_A is
begin
   invec : TYPE_B;
   invec = vec_b;
   vec_a = a(invec);
end
```

- **compute-vector method**: The computation of a new state vector in **executive** style is distinct from returning its result. To compute **`A`**'s new vector, we must first compute the state vector of the layer immediately below **`A`** (by calling its **compute-vector** method). We then generate the call "**`READ_A;`**". For equation **`E`**, the calls that would be produced by invoking the **compute-vector** method of **`A`** is:

```
READ_C;
READ_B;
READ_A;
```

This sequence of calls is included in the task-loop of **`Main`** of *Figure 4*.

Note when the type equation **`E`** is created, one is actually composing *metaprogramming* implementations for each of **`E`**'s components. When the generator executes **`E`**, it produces/generates the executive source code of *Figure 4*. In the next section, we will show how a layer-style adaptor can be written.

5.3 A layer-style adaptor

A metaprogramming adaptor intercepts method calls for code generation and replaces them with different calls. Here are the refinements for a **layer**-style adaptor called **`layer`**:[4]

- **global-variable method**: To make component **`A`** appear to be in a layered architectural style, **`A`** will have no global variables. When the **global-variable** method of the **`layer`** adaptor is called, a dispatch to the **global-variable** method of the component immediately below **`A`** is

4. Note that **`x = layer[x]`** is an architectural style identity.

called (thereby skipping the call of `A`'s **global-variable** method). So, the variable declarations generated for the equation `E' = layer[A[B[C]]]` would be:

```
vec_b : TYPE_B;
vec_c : TYPE_C;
```

That is, components `B` and `C` are still in executive style (and thus have global variables), but `A` is not.

- **get-current-vector method**: To obtain the current vector in layered style, `A` would output the assignment statement "`invec = READ_A;`", where `READ_A` is a function that returns `A`'s current state vector.
- **compute-vector method**: The computation of a new state vector in layered style occurs whenever its `READ_A` function is called. Thus, the **compute-vector** method of a **layer** adaptor generates no code and has a null body. An example of this method will be given shortly.
- **interface-generation method**: `A`'s read-vector method in layered style involves the generation of a parameterless function that returns `A`'s state vector:

```
function READ_A return TYPE_A is
begin
   invec : TYPE_B;
    --- invoke compute-vector
    --- set invec to appropriate value
   return a(invec);
end
```

The above function is incomplete, because the computation of the state vector from the component beneath `A` must be performed and local variable `invec` must be initialized. The code for the latter is produced by calling the **compute-vector** method, and the code for the latter is produced by calling the **get-current-vector** method of the component beneath `A`. As an example, the code generated for the equation `E' = Main[layer[A[B[C]]]` would be:

```
function READ_A return TYPE_A is
begin
   invec : TYPE_B;
   READ_C;         --- compute-vector before referencing
   READ_B;
   invec = vec_b; --- variable invec equals vec_b
   return a(invec);
end
```

5.3.1 A task-style adaptor

A **task**-style adaptor (called **`task`**) would have the following methods:

- **global-variable method**: There are no global variables in **task** architectural styles. The **global-variable** method of a task adaptor simply returns the result of the **global-variable** method of the component beneath **`A`**.
- **get-current-vector method**: To obtain the current vector in task-style, **`A`** would output the assignment statement "**`TASK_A.READ_A(invec);`**", which assigns variable **`invec`** a value via a task call.
- **compute-vector method**: As with the layer-style adaptor, the computation of a new state vector in task-style occurs whenever its task read-vector method is called. Thus, the compute-vector method of a layer adaptor has a null body. An example will be given shortly.
- **interface-generation method**: **`A`**'s read-vector method in task style generates an Ada task:[5]

```
task TASK_A is
   entry READ_A( vec_a : out TYPE_A );
   ...
end;
task body TASK_A
begin
   loop
      accept READ_A( vec_a : out TYPE_A ) do
         invec : TYPE_B;
          --- invoke compute-vector
          --- set invec to appropriate value
         vec_a = a(invec);
      end;
      ...
   end loop
end
```

As an example, the code generated for the equation **`E' = Main[task[A[B[C]]]]`** would be:

```
task TASK_A is
   entry READ_A( vec_a : out TYPE_A );
   ...
end;
```

5. Readers may note that the Ada **`uses`** clause specifies tasks that can be called from within a task. The list of such tasks could be produced by an additional method — **`uses-tasks`** method — that all components would need to implement.

```
task body TA
begin
   loop
      accept READ_A( vec_a : out TYPE_A ) do
         invec : TYPE_B;
         READ_C;
         READ_B;
         invec = vec_b;
         vec_a = a(invec);
      end;
      ...
   end loop
end
```

5.3.2 Recap

Given the above model of components and adaptors, the type equations for *Figures 4-6*, which are equivalent to equation `E`, are:

```
Figure4 = Main[A[B[C]]];
Figure5 = Main[layer[A[layer[B[layer[C]]]]]];
Figure6 = Main[task[A[task[B[task[C]]]]]];
```

It is not difficult to imagine that metaprogramming adaptors for other architectural styles — such as table dispatching, file filters, and Weaves (Gorlick and Razouk, 1991) — can be created by following the above approach. It is also not difficult to see that different architectural styles can be intermixed within the same type equation. Thus, a version of `E` that implements `A` as a task, `B` in layered style, and `C` in executive style would be `E* = Main[task[A[layered[B[C]]]]]`. The source that would be generated from this equation is shown in the Appendix.

Readers may have noticed that more compact code could be generated in our examples. For example, the `invec` variable could easily be removed from many of our generated procedures. While this is a trivial optimization, it is symptomatic of inefficiencies that can arise in metaprogramming implementations of components and adaptors. Optimizations requiring code movement and variable elimination are extremely difficult to express in metaprograms. If such optimizations are crucial for producing efficient code, then rather than implementing components and adaptors as metaprograms, a better way would be to implement them as rule-sets of program transformation systems (where such optimizations are possible and can be expressed easily). Again, this is possible in a GenVoca model because the basic model remains unchanged; it is only the implementation the generator (and the domain model components) that are affected.

6 CONCLUSIONS

Product-line architectures are becoming progressively more important. Isolated designs of individual software products are being replaced with designs for product-lines that amortize the cost of both building and designing families of related products. A critical aspect of product-line designs involves architectural styles. Different applications of a product family may require the use of different styles as the basis of component communication. Simple and comprehensible models of product lines demand the interchangeability of architectural styles.

In this paper, we have explored the relationship of architectural styles and GenVoca models. Our approach outlined first steps towards viewing architectural styles as adaptors (Gamma, et al., 1994). Since GenVoca represents applications as equations (i.e., compositions of components), adaptors have a particularly appealing representation as algebraic identities. That is, the ability to replace one architectural style with another is elegantly expressed by rewriting an equation using an algebraic identity. Moreover, the central concept of GenVoca — namely building blocks of product line architectures are refinements — was unaffected. Both components and adaptors are examples of refinements.

We presented deliberately simplified examples of avionics software that were coded in different architectural styles. We explained how metaprogramming implementations of components and adaptors could achieve the effect of synthesizing these examples through component composition. This demonstrated the important effect that adaptors and components could be designed to be orthogonal to each other, thereby admitting a mix-and-match capability that is both desirable and characteristic of GenVoca designs.

Most approaches to architectural styles do not adopt the wholistic view that we have taken, namely that one designs components and adaptors to work together to achieve a mix-and-match capability. Typically approaches begin with pre-existing components; the task is to develop tools that will alter the architectural styles by means of component unwrapping and/or rewrapping. While this approach will achieve success, we believe that an approach that integrates component and adaptor designs will yield stronger results and less fragile tools in developing product line architectures of the future.

REFERENCES

Batory, D. and O'Malley, S. (1992), "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM TOSEM*, pp.355-398.

Batory, D. and Coglianese, L. (1993), "Techniques for Software System Synthesis in ADAGE", ADAGE-UT-93-05, Loral Federal Systems Division.

Batory, D., et al. (1993), "Scalable Software Libraries", *Proc. ACM SIGSOFT.*

Batory, D. and Smaragdakis, Y. (1995), "Architectural Styles and Adage", UT-ADAGE-95-02, Loral Federal Systems Division.

Batory, D. and Geraci, B.J. (1997), "Composition Validation and Subjectivity in GenVoca Generators", *IEEE Transactions on Software Engineering*, pp.67-82.

Biggerstaff, T. (1994), "The Library Scaling Problem and the Limits of Concrete Component Reuse", *International Conference on Software Reuse*, pp.102-109.

Blaine, L. and Goldberg, A. (1991), "DTRE - A Semi-Automatic Transformation System", in *Constructing Programs from Specifications*, Elsevier Science Publishers.

Coglianese, L. and Szymanski, R. (1993), "DSSA-ADAGE: An Environment for Architecture-based Avionics Development", *Proc. AGARD.*

Cohen, S., et al. (1995), "Models for Domains and Architectures: A Prescription for Systematic Software Reuse", *AIAA Computing in Aerospace.*

DeLine, R. (1996), "Toward User-Defined Element Types and Architectural styles", position paper in *Second International Software Architecture Workshop*, pp.47-49.

Dijkstra, E.W. (1968), "The Structure of THE Multiprogramming System", *Communications of ACM*, pp.341-346.

Gamma, E.; Helm, R. ; Johnson, R. and Vlissides, J. (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Garlan, D., et al. (1994), "Exploiting Style in Architectural Design Environments", *ACM SIGSOFT*, pp.175-188.

Garlan, D., et al (1995), "Architectural Mismatch or Why It's Hard to Build Systems out of Existing Parts", *Int. Conf. on Softw. Eng.*

Gorlick, M.M. and Razouk, R.R. (1991), "Using Weaves for Software Construction and Analysis", *Int. Conf. Softw. Eng.*, 23-34.

Harrison, W. and Ossher, H. (1993), "Subject-Oriented Programming (A Critique of Pure Objects)", *OOPSLA*, pp.411-427.

Heidemann, J. and Popek, G. (1993), "File System Development with Stackable Layers", *ACM TOCS.*

Hutchinson, N. and Peterson, L. (1991), "The *x*-kernel: An Architecture for Implementing Network Protocols", *IEEE TSE*, pp.64-76.

do Prado Leite, J.C.S. , et al. (1994), "Draco-PUC: A Technology Assembly for Domain-Oriented Software Development", *International Conference on Software Reuse*, 94-101.

McIlroy, M. D. (1968), "Mass-Produced Software Components", In *Proceedings of the NATO Conference on Software Engineering.*

Neighbors, J. (1980), "Software Construction Using Components", Ph.D. Thesis, ICS-TR-160, University of California at Irvine.

Parnas, D. L. (1976), "On the Design and Development of Program Families", *IEEE Transactions on Software Engineering*.

Perry, D. E. and Wolf, A. L. (1992), "Foundations for the Study of Software Architecture", *Software Engineering Notes*, 17(4).

Shaw, M. and Clements, P. (1997), "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems", Proc. *COMPSAC97, 21st International Computer Software and Applications Conference*, pp. 6-13.

Smaragdakis, Y. and Batory, D. (1997), "DiSTiL: a Transformation Library for Data Structures", *Conference on Domain Specific Languages* (DSL '97).

Smaragdakis, Y. and Batory, D. (1998), "Implementing Layered Designs with Mixin Layers", *European Conference on Object-Oriented Programming*.

Weiss, D.M. (1990), *Synthesis Operational Scenarios*, Technical Report 90038-N. Version 1.00.01, Software Productivity Consortium.

APPENDIX - SOURCE FOR `Main[task[A[layered[B[C]]]]]`

```
-- global state vectors
vec_c : TYPE_C;

procedure READ_C is
begin
   vec_c = c();
end;

function READ_B return TYPE_B is
begin
   invec : TYPE_B;
   READ_C;
   invec = vec_c;
   return b(invec);
end;

task TASK_A is
   entry READ_A( vec_a : out TYPE_A );
   ...
end;
task body TASK_A
begin
   loop
      accept READ_A( vec_a : out TYPE_A ) do
         invec : TYPE_B;
         invec = READ_B();
         vec_a = a(invec);
      end;
      ...
   end loop
end

-- main task

task body MAIN
use TASK_A is
begin
   x : integer;
   invec : TYPE_A;
   loop
      TASK_A.READ_A(invec);
      -- compute time x till next cycle;
      delay x;
   end loop
end;
```

Attribute-Based Architecture Styles

Mark H. Klein, Rick Kazman, Len Bass, Jeromy Carriere, Mario Barbacci, and Howard Lipson
Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213
{mk, rkazman, ljb, sjc, mrb, hfl}@sei.cmu.edu

Key words: Architecture, architecture styles, quality attributes

Abstract: Architectural styles have enjoyed widespread popularity in the past few years, and for good reason: they represent the distilled wisdom of many experienced architects and guide less experienced architects in designing their architectures. However, architectural styles employ qualitative reasoning to motivate when and under what conditions they should be used. In this paper we present the concept of an ABAS (Attribute-Based Architectural Style) which includes a set of components and connectors along with their topology, but which adds to this a quality attribute specific model that provides a method of reasoning about the behavior of component types that interact in the defined pattern. We will define ABASs in this paper, show how they are used, and argue for why this extension to the notion of architectural style is an important step toward creating a true engineering discipline of architectural design.

1. INTRODUCTION

An architectural style (as defined by Shaw and Garlan (Shaw and Garlan, 1996) and elaborated on by others (Buschmann, et al., 1996)) includes a description of component types and their topology, a description of the pattern of data and control interaction among the components and an informal description of the benefits and drawbacks of using that style. Architectural styles are important since they differentiate classes of designs by offering experiential evidence of how each class has been used along with qualitative reasoning to explain why each class has certain properties."Use the pipe and filter style when reuse is desired and performance is not a top priority" is an example of the type of description that is a portion of the definition of the pipe and filter style. The purpose of this paper is to move the notion of architectural styles toward having the reasoning (whether qualitative or quantitative) based on quality attribute-specific models. We call these enhanced architectural styles, Attribute-Based Architecture Styles (ABASs) and we view them as the next generation in the development of architectural styles.

We define an ABAS as a triple

1. the topology of component types and a description of the pattern of data and control interaction among the components (as in the standard definition),
2. a quality attribute specific model that provides a method of reasoning about the behavior of component types that interact in the defined pattern, and
3. the reasoning that results from applying the attribute specific model to the interacting component types.

Thus, to further use the pipe-and-filter example, a pipe-and-filter *performance* ABAS would be one that has a description of what it means to be a pipe or a filter and how they would legally be connected, a queuing model of the pipe-and-filter topology together with rules to instantiate the model, and the results of solving the resulting queuing model under varying sets of assumptions.

Software architecture styles are useful during both design and analysis. Styles are useful during design because the software architect can choose a style based on an understanding of the desired quality goals of the system under construction. The goal of those cataloguing architectural styles (Buschmann, et al., 1996) is to provide a handbook that the software architect can use as a reference to have design options with known qualities from which to choose.

In this paper, we make two points. The first (rather obvious) point is that architectural styles are also useful in analysis. When analyzing a system, the recognition of the use of pipe and filter, for example, leads to questions about how performance is handled and about the assumptions that the filters make that might impact their reuse. The second point (somewhat less obvious) is that when considering architectural styles as analysis tools, focussing on particular quality attributes (McCall, 1994) leads to the ability to attach known analytic models for these attributes to the architecture being analyzed. This in turn leads to the ability to *predict* the effect of particular architectural decisions and changes to the architecture. Thus, instead of the designer having vague guidance about a particular style's effect on performance, the designer is given a model, its analysis, and its explicit connection to aspects of the architectural style so that the designer can answer questions such as “What is the effect on performance of moving a particular piece of functionality from one component to another within a pipe and file based architectural design?”

In the remainder of this paper, we discuss the roots of the ABAS concept, the pieces of an ABAS, the types of attribute models that exist and how they would be used in constructing an ABAS, an extended pedagogical example, and an example drawn from our experience using ABASs in architectural analysis that shows how ABASs work in practice.

2. MOTIVATIONS

The motivation for ABASs comes from three different sources:

1. architectural styles, such as those catalogued by Shaw and Garlan in (Shaw and Garlan, 1996) and by Buschmann et al in (Buschmann, et al., 1996)
2. analytic models of quality attributes, such as rate monotonic analysis for performance (Klein, et al., 1993) or Markov models for availability
3. architecture evaluation questionnaires, such as those used by AT&T (Maranzano, 1993)

ABASs are a kind of architectural style, and hence they build squarely upon the foundational work of Shaw and Garlan, as well as the similar work of the design patterns community (Gamma, et al., 1994). However, in each of these cases, the kinds of reasoning that the architectural styles support is heuristic. For example, in describing the layered style, Shaw and Garlan write “if a system can logically be structured in layers, considerations of performance may require closer coupling between logically high-level functions and their lower-level implementations”. While this is important information for the designer who is considering using this style, it does not give the designer a principled way of understanding when a specific number and organization of layers will cause a performance problem. The answer to this dilemma lies in our second influence, analytic models of quality attributes.

Mature analytic models exist for several quality attributes that are of central concern to complex software systems, such as performance, reliability and, to a lesser extent, security. These models not only provide a way to establish a more precise understanding of, for example, “considerations of performance”, but also can allow the analyst to associate particular measurable performance criteria with architectural choices. This gives the designer a way to rigorously experiment with, and plan for, architectural quality requirements.

However, analytic models are typically quite general and it requires a substantial amount of training to be able to use them effectively. Frequently, when we perform architectural evaluations, we need to be able to assess a design's effectiveness and risk within a 2 or 3 day period. This leads to the our third motivation. A proven technique for aiding in risk assessments of architectures, first used widely by AT&T's software architecture validation exercises (Maranzano, 1993), is a questionnaire or checklist. A proven set of questions can help organize the line of reasoning and investigation into an architecture, and can provide a first insight into problem areas. These questions can be a first approximation for an analysis, and can lead the analyst in probing the architecture.

3. MODELING ARCHITECTURAL DECISIONS USING AN ABAS

One of the reasons for focusing attention on an architecture is to highlight and analyze critical early design decisions. Translating these decisions into some modelling framework that supports predictive reasoning at the architecture level is key for attaining the potential benefits of focusing on the architecture. The structure of an ABAS reflects this goal of *mapping* an architecture style onto an attribute-modelling framework. This notion is represented by Figure 1.

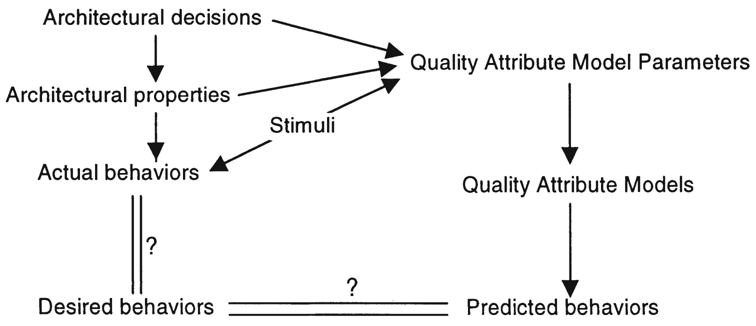


Figure 1. Mapping architectural models to attribute models

The left side of the figure says that architectural decisions directly and/or indirectly affect the behavior ultimately manifested by the architecture. That is obvious, but what is less clear is how to characterize those behaviors and to understand how they compare with the desired behaviors. For example, allocating functionality to a collection of processes (a subset of the architectural decisions that a designer will make) that are in turn allocated to processors (more architectural decisions) result in a set of process execution times (that is, architectural properties). Architectural properties in conjunction with stimuli such as message arrival rates ultimately lead to the performance behavior that will be exhibited. The actual behavior of the system is unknowable without constructing the system and so we use models of the behavior as a method of characterizing the actual behaviors.

We can, of course, "hope" that the actual behavior will satisfy the desired behavior, but there is no way to know unless some type of model is used. The architecture abstraction needs to be mapped to some other abstraction that is more supportive of reasoning. For example, if the goal is to reason about reliability, the salient features of the architecture (such as redundancy) need to be mapped onto reliability models (such as Markov models). At this point the behaviors predicted by the models can be compared with the desired behaviors. Such reasoning can become the basis for comparing architectures and for making decisions regarding the form of the final software architecture.

3.1 The Structure of an ABAS

We define an ABAS to have five parts:

1. **Problem description** - describes the design problem that the ABAS is intended to solve, including the quality attribute of interest, the context of use, constraints, and relevant attribute-specific requirements.
2. **Quality attribute measures** - a condensation of what was discussed in the problem description, but in specific terms pertinent to the *measurable aspects* of the quality attribute model. This includes a discussion of *stimuli*: events that cause the architecture to respond or change.
3. **Architectural style** - a description of the architectural style in terms of component, connections, properties of the components and connections, and patterns of data and control interactions.
4. **Quality attribute parameters** - a condensation of what was discussed in the architectural style section but in specific terms relevant to the *parameters* of the quality attribute model.
5. **Analysis** - a description of how the quality attribute models are *formally related* to the elements of the architectural style and the conclusions about architectural behavior that are drawn via the models.

Note that these parts rely on a description of the architectural style and on a description of a quality attribute. Describing quality attributes is discussed in the next section.

4. QUALITY ATTRIBUTE MODELS PARAMETERS

To assess an architecture for adherence to quality requirements, those requirements need to be expressed in terms that are measurable or at least observable. We call these the *quality attribute measures*. These parameters depend on properties of the architecture, called *quality attribute parameters*, and on the *stimuli***.** The quality attribute parameters are the adjustable parameters of the architecture that determine whether the dependent parameters will satisfy the quality requirements. Stimuli are the events to which the architecture will have to respond.

Consider performance: performance is concerned with timeliness, usually measured as either latency or throughput. Therefore two **quality attribute measures** are:

- **Latency** - time from the occurrence of an event until the response to that event is complete; expressed in units of time.
- **Throughput** - the rate at which the system can respond to events; expressed in terms of transactions (or responses) per unit time.

The **stimuli** are changes of state to which the architecture must respond. For performance the arrival pattern of an event is important. It can be one of:

- **Periodic** - there is a fixed interval between event arrivals.
- **Sporadic** - there is a bound on how short the interval between arrivals can be.
- **Stochastic** - arrivals can be described probabilistically.

When a stimulus occurs, the system responds to it by using its resources to carry out computations or transmit data. Multiple concurrent stimulus responses require an arbitration or scheduling policy to resolve conflicting requests. Thus we think of performance-related architectural parameters in terms of the resources that are needed, the policies for allocating resources, and properties of how the resources are requested and used. Therefore **quality attribute parameters** include:

- **Resource characteristics** - include the type of resource such as CPU or network and characteristics such as processor speed or network bandwidth.
- **Resource scheduling policy** - includes CPU scheduling, CPU allocation, and bus and network arbitration; and queuing policies.
- **Resource usage** - includes the priority of processes and messages, preemptability of response and magnitude of use such as execution time.

ABASs map a characterization of architectural properties onto quality attribute parameters, and then map (via modelling) quality attribute parameters and stimuli onto predicted behaviors. Models such as those for scheduling and queuing provide the basis for relating quality attribute parameters (such as queuing policies and execution time estimates) to quality attribute measures (such as latency and throughput). Some parameters such as execution time might not be easily quantifiable at the architecture level. In this case execution time budgets can be assigned, which then become derived requirements for fleshing out the details of the component. This is further illustrated in the next section in which we discuss a sample ABAS for reliability.

For other attributes such as *reusability* or *modifiability,* where there are no universal quality attribute measures, scenarios can used to provide context dependent measures. (Kazman, et al., 1996)

5. ABASs

We will illustrate the notion of an ABAS by an example. We are currently collecting, documenting, and testing many such examples in the hope of creating an engineering handbook of ABASs. The example given here uses a form of redundancy known as analytic redundancy as a means of achieving high levels of availability. First, we will lay out a portion of the relevant attribute model[1].

5.1 Reliability/Availability Attribute Model

Reliability is usually measured in terms of mean time to failure (MTTF). Availability is usually measured in terms of the long-run fraction of time that a system is working. Component failures (and faults)[2], and repair (or recovery) are the stimuli of concern. Architecture parameters include fault detection and fault containment and recovery strategies. An attribute model for reliability/availability looks as follows[3]:

Quality attribute measures

- **Steady state availability** — fraction of time that the system is working (that is, not in a failed state)
- **Reliability —** usually measured in terms of mean time to failure
- **Faults detectable** — passive failures (detectable via time-out mechanisms), active failures, timing failures, semantic failures

[1]An attribute model does not have to be developed for every ABAS. Only one attribute model is needed per attribute and it is then applied to all ABASs for which that specific attribute is relevant.

[2]In this paper we do not distinguish between failures and faults.

[3]This is not meant to be a complete attribute model, but rather one that focuses attention on architectural decisions.

Stimuli -- characterized in terms of the types of failures and repairs and their rates.

Quality attribute parameters
- **Detection -** mechanisms for detection of failures such as voting, post-condition checking, and deadline detection
- **Recovery -** mechanisms for recovering from failure including forward and backward recovery mechanism
- **Modes** - A system can operate in various degraded modes and the availability/reliability of each mode of operation have to be calculated separately.

5.2 Simplex ABAS

We will now describe the documentation that accompanies an ABAS by means of an example of a particular ABAS, called Simplex.

5.2.1 Problem Description

The purpose of this section is to describe the architectural design problem being addressed or in other words the goals of the architecture.[4]

The Simplex (Sha, et al.) ABAS focuses on the problem of software reliability in control systems. In particular, Simplex addresses the problem of tolerating software faults introduced as a consequence of upgrading control algorithms. Simplex also addresses the problem how to take advantage of redundancy to increase reliability while avoiding "common mode" software failures.

To illustrate the problem consider, "the update paradox", as described in (Sha, et al., 1996). Consider the case in which a component is replicated to ensure its reliability. Each replica performs its calculations and sends its results to a voter. If the results do not agree (to within a specified tolerance), the voter "votes for the majority".

Let's say that a key algorithm is updated which will yield a different output value than the older algorithm. Here's the paradox: if the new algorithm is placed in a minority of the replicated components then it will be voted out and have no effect; if it's placed into a majority of the replicated components and is faulty, the bad output will used.

There are two problems highlighted by the upgrade paradox. First of all, even components that have been implemented by different groups and hence have different implementations can suffer from common mode failures. Hence the first problem is, how to introduce redundancy to ensure the proper

[4]The text in italics in this section is commentary for the reader, and is not part of the ABAS.

level of reliability/availability without introducing common mode failures? The second problem is, how do you upgrade a system without compromising its reliability/availability?

5.2.2 Quality Attribute measures and Stimuli

Based on the desired architectural behavior, the stimuli of the reliability/availability attribute model, and the problem description there are set of specific issues that should be highlighted. These issues are raised in this section. A checklist of such issues would include those that follow.

The availability/reliability issues of concern for this ABAS are:

- What types of faults need to be tolerated by the architecture
- What the levels of (degraded) service are
- What the reliability/availability is for each level of service

Types of faults: the goal of this architecture is to handle timing faults (e.g., timing overruns), semantic faults (wrong output values) and system faults (such as memory overruns due to bad pointers).

Reliability of service levels: There is a specified desired level of availability for the upgraded or higher performance level of service and specified level of reliability for the baseline level of service.

5.2.3 Architectural Style

This section starts by identifying the relationship between this ABAS and other similar ABASs. In this case the Simplex ABAS is an instance of a more general pattern.

Simplex is an architectural style which belongs to a general family of reliability styles that could be called redundancy styles. The general pattern for a redundancy style is shown in Figure 2 below. The pattern, from a reliability point of view, consists of multiple redundant components. Data flows into one or more redundant components, which then send their output to another component (or possibly components) responsible for detecting failures, switching to a working component and possibly initiating recovery of the failed component.

The Simplex style, as shown in Figure 3, is an instance of the redundancy style in which the redundant components are processes. The components don't necessarily receive the same input or generate the same output. Moreover, the components are not all peers. The components are *analytically redundant*, meaning they are redundant with respect to the general effect their output has in controlling their environment, but not necessarily redundant in the algorithms used or the output produced. [5]

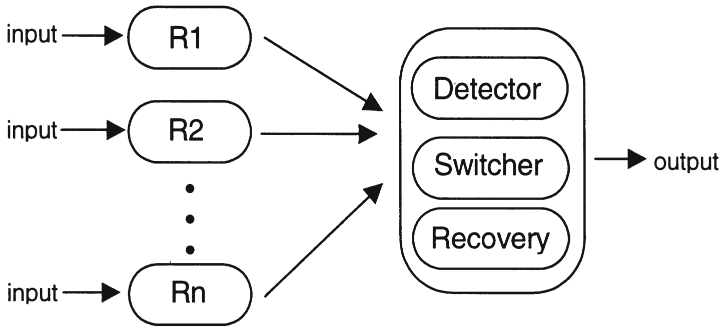


Figure 2: A redundancy-specific architectural style

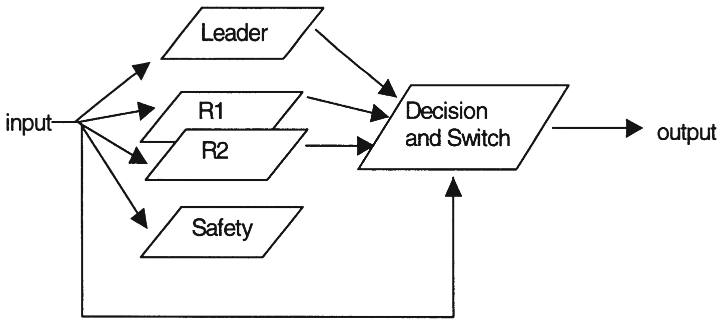


Figure 3: The Simplex architectural style

The "leader" component, the other redundant components (R1 and R2) and the "safety" component are analytically redundant. The "leader" is typically the upgraded version of a critical component. All components execute concurrently. The leader's output is used if it passes the acceptance test applied by the decision and switch unit. The acceptance test is based on a model of the controlled environment and the ability of the safety

[5]You can think of the relationship between power steering and mechanical steering as analytic redundant. Both mechanisms have the same effect on the environment, that is, they change the direction of the wheels, but the mechanisms used, the output produced, and their performance are all different.

component to recover from actions of the other components. If the leader doesn't pass this test a new leader is picked (either R1 or R2). The "safety" component is a simple, highly reliable analytically redundant component that is used as a last resort. The safety might be used to affect a recovery to the point where one of the other (more able) components can once again take over. Note that the decision and switch component receives a copy of the input and uses it as a basis for performing its acceptance test.

The Simplex style assumes that mechanisms exist to bound the execution time of the components, thereby preventing timing overruns. Another (related) style would address performance issues. The Simplex style also assumes that the concurrent units are processes with address space protection thereby preventing the propagation of system faults such as memory overruns.

5.2.4 Quality attribute parameters

Based on the architecture parameters of the reliability/availability attribute model and on the pattern of interactions, there are set of specific issues that should be raised to refine the pattern.

The quality attribute parameters of concern for this ABAS are:

- Analytic redundancy (possibly different input; different implementation; possibly different output) is the form of *redundancy*
- A leadership based "*voting*" mechanism is used.
- Estimates are needed for *failure rates* and *repair rates* of the various components. We assume that the failure rates for the decision unit and the safety component are very low in comparison to the failures rates of the other components.

5.2.5 Analysis

This section ties together the preceding sections. It discusses how to use the architectural decisions and properties and the stimuli to model the architectural behavior.

To model the availability of this style you have to make estimates of the failure rates and repair rates of the components to calculate the availability of the system. Reliability growth models can be used for obtaining estimates. In addition, it can be very illustrative to compare one architecture style to another simply by making assumptions about the various failure and repair rates. This is the approach we will use.

The first step in this section is to map the architectural decisions and properties into a quantitative or qualitative model that helps you to predict the architectural behavior.

Modelling a Simpler Problem

Before discussing the analysis of the Simplex style we'll first take a look at a similar style, the *majority voting* style.[6] This is the style that we used in the problem description to illustrate the update paradox. For this style there are three redundant components[7]. At least 2 or the 3 components must produce results that agree, otherwise the system has failed. When the system is working (in this case controlling some aspect of its environment, for example, the trajectory of a missile or the temperature and pressure of a chemical process) it is performing at a constant level of service. The system can be in one of three states: 3 working components, 2 working components, or failed. If F failures per year occur and a component repair takes on the average 1/R years then the Markov model shown in Figure 4 can be used to calculate the availability (that is, the proportion of time that the system is not in the failed state).

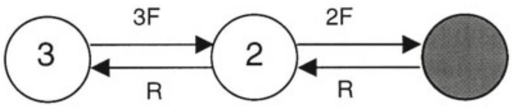


Figure 4: A Markov model for majority voting

The representation of a Markov model in Figure 4 can be viewed as a state diagram. State "3" represents the state in which 3 components are working, state "2" represents the state in which 2 components are working and the grey state is the failed state. The transition arrows are labelled with failure (F) and repair (R) rates. Since each component fails independently with an average rate of F, 3 components fail with an average fail rate of 3F and hence the label for the transition from state "3" to state "2".

The steady state solution of the Markov model yields the long-term proportion of time that the system is in each state. Therefore the availability of the majority voting case is the proportion of time in which the system is in state "3" or state "2", and hence not in the failure state.

More information about Markov models can be found in standard texts on probability. Our goal is to illustrate the mapping from architectural parameters to a predictive model and to show how the model provides the motivation for the characterization of the ABAS. In this case the predictive

[6] The majority voting style would probably have its own entry in a handbook of ABASs and be referenced in the Simplex ABAS.

[7]This is known as Trimodular redundancy (TMR). However, majority voting is not restricted to 3 components.

model is a mathematical model. In other ABASs qualitative reasoning techniques might also be used. For this case we use the model to gain an understanding of how the availability varies as a function of the assumed failure and repair rates, not to get absolute availability estimates. The trends of the majority voting style will then be compared with Simplex style.

Modeling Simplex

The Simplex style achieves relatively high levels of availability of the high performance (e.g., a very precise algorithm) variant by using a highly reliable but lower performing (e.g., a less accurate algorithm) variant to recover from faults. To illustrate the concept consider a system with two redundant controllers (R1 and R2), a safety controller, and a monitoring and decision unit. The Simplex style preserves the total number of active components, but allocates functions to components differently depending on their states, and hence the components have different failure properties. The Markov model for this style is shown in Figure 5.

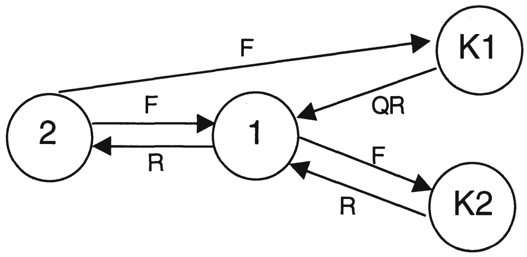


Figure 5: Markov model for Simplex

The system starts in state "2" with two high performance controllers, the outputs of which are compared. If they agree we assume that they're correct (that is, we assume no common mode failure, but rather random failures). If they disagree, one is picked. If the right one is picked the model transitions from state "2" to state "1". If the wrong one is picked the model transitions from state "2" to state "K1", where K1 stands for the a state in which the safety component becomes active. Since one of the high performance controllers continues to work, the transition from "K1" to "1" is relatively quick and thus has a quick repair (QR) rate. We assume that QR=n*R, for some n greater than 1. If a failure occurs while in state "1", the system also transitions to the safety controller, but in this case the repair rate is that of a "normal" repair (i.e. a software or hardware fix).

The final objective is to gain insight into the architecture by using the model as a basis of reasoning.

A key to the availability properties of this style is the relatively quick repair rate (QR) from state “K1” to state “1”. To see this imagine that QR is so quick that virtually no time is spent in state “K1”. In this case the model in Figure 5 closely approximates the model in Figure 6, below. The availability properties of the model shown in Figure 6 are better than for majority voting (shown in Figure 4) due to the higher transition rates for majority voting. The higher transition rates for majority voting are a consequence of needing a majority of the redundant components to agree in order to detect a failure, whereas this style uses a semantic check of the output for failure detection.

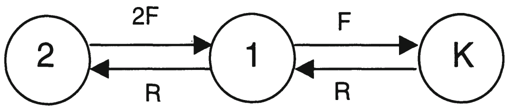


Figure 6: An approximate Markov model for Simplex

6. USING ABASs

We have applied ABASs on a real-world system during an enactment of the Architecture Tradeoff Analysis Method (ATAM) (Kazman, et al., 1998). During the course of the architectural analysis, ABASs relevant to several properties (availability, performance, and modifiability) were applied to aid in the understanding of the system and the consideration of design alternatives.

The system being evaluated—which we will call LAOB (Leader And One Backup) here[8]—comprises a collection of independently operating nodes (computers), communicating via a radio network, with a single node acting as leader. The leader has the responsibility to plan the activities of the other nodes. To perform this planning, it must accumulate and maintain data concerning the states of the other nodes.

Because the availability of the system is critical, we used a reliability ABAS to map the quality attribute parameters (i.e. the architectural decisions, such as the mechanisms for detection and recovery) and the predicted stimuli (e.g. failure of a node) onto the quality attribute measures (i.e. the predicted behavior) of the system via a reliability model. The resulting analysis was used to understand how well the system will meet its

[8]The actual name, developing organization, and details of this application are proprietary, but their suppression does not affect the analysis.

availability goals and to inform decisions for refining the architecture. In particular, by looking at the system via ABASs, we were able to determine that its reliability had not been adequately addressed in either requirements or implementation.

Quality Attribute Measures: Based on the reliability/availability attribute model presented in Section 5.1, the quality attribute measure of interest for this ABAS is its steady-state availability.

Stimuli: The stimuli of interest for this ABAS are hardware or software failures of the nodes.

Structure: The structure of the ABAS is shown in Figure 7. Communication takes place exclusively between the leader and the other nodes (i.e. the nodes do not communicate with each other). If the leader fails, a node pre-designated as a backup must reconfigure to take on the planning responsibilities of the leader and must acquire any additional data it needs to begin performing the leadership responsibilities. Also, another node must be identified to act as the new backup.

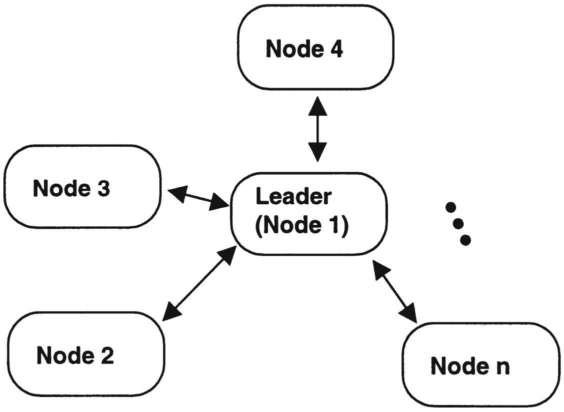


Figure 7: The ABAS-relevant structure of the LAOB system

Quality Attribute Parameters: The quality attribute parameters of interest in this style are:

– The mechanism used for detecting the failure of the leader: In the LAOB system, the lack of communication between the backup and the leader signals that the leader has failed.

- The mechanism for recovering system operation: When the leader fails, a designated backup takes over. The backup must acquire whatever data it requires to begin acting as leader and reconfigure itself.
- The failure and repair rates for the leader and the other nodes: The failure rates for the leader and the other nodes may be different as the leader has different responsibilities and is executing different software. The repair rates will need to include the time required for a node to take over as leader and the time required for a node to take over as backup. When we speak of repair, we are referring to the repair of the system, returning it to a functioning state from a non-functioning one. Individual nodes that have failed do not get repaired during the execution of the system.

Analysis: From our generic reliability ABAS we know that we can model this system using a Markov model. Figure 8 shows the Markov model for a three node system (it is easily generalized to more nodes). Each state in the model is labelled with a triple: (number of leaders active/number of backups active/total number of nodes active). F_l is the failure rate of a leader, F_b is the failure rate for the backup node, F_o is the failure rate for another node, R_l is the repair rate for the leader (i.e. the reciprocal of the time taken to transform a backup into the leader) and R_b is the repair rate for a backup (i.e. the reciprocal of the time taken to transform another node into a backup). The model makes the assumption that the transformation of the backup into leader and the transformation of another node into backup are sequential.

The steady-state availability can now be computed as the probability of the system being in a state in which a leader is active (four of the eight states). Based on expected failure and repair rates, the model can be used to understand how well the system will meet its availability goal.

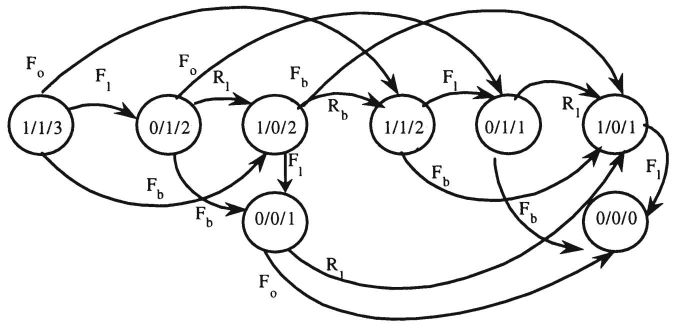


Figure 8: A Markov model for a reliability ABAS

Reasoning about the system in the context of the reliability ABAS led us to a consideration of other architectural alternatives for the LAOB system. The primary alternative considered was the use of multiple backups to turn the LAOB into a LAMB (Leader And Many Backups), where each of the backups would maintain the state necessary to quickly take over upon failure of the leader.

This alternative will result in better availability due to a reduced repair time, but at a cost of higher utilization of the radio network. Since the radio network had a relatively low bandwidth, this was not a trivial consideration: keeping additional backups informed of the state of the leader meant additional transmissions and retransmissions. The performance issues for the LAOB/LAMB system alternatives were considered using a separate ABAS, one for communicating processes, and the confluence of these two ABASs identified an architectural tradeoff, since higher levels of availability meant higher utilization of the network.

The purpose of this example is not to present the design decisions made for this system, or even the details of the analysis, for they are not the point of this paper. The point is that a consideration of ABASs led us to ask questions of the system: reliability ABASs made us ask questions about failures and recovery of components and their effects on the predicted level of system availability; performance ABASs made us ask questions about resource characteristics and resource utilization and their effects on the predicted level of system response times. Using these models, we could play with different architectural alternatives, constantly gauging the performance of these alternatives against the system's requirements. For example, we could explore versions of the LAMB system with varying numbers of backups and with different strategies for keeping them synchronized with the leader. Strategies include:

1. The backups could be passive recipients of updates, not worrying about any missed information until they are called upon to become the leader. In this case they would not be guaranteed of being true functional replicas of the leader.
2. They could be active recipients, requesting re-sends of any missed packets (they could identify missed packets via noting holes in the packet number sequence, for example). In this case they will be functional replicas of the leader most of the time, but at the cost of additional communication with the leader.
3. A single backup could be an active recipient and all other backups could be passive recipients. When the primary backup was called upon to become the leader, it would designate a new primary backup and negotiate with it to provide it with any missed packets, at the cost of additional communication at switchover time.

The various strategies each have different availability and performance implications—different bandwidth requirements, different time to repair, different probabilities of failure—and these can be modelled analytically before committing to one strategy for prototyping or implementation. Perhaps more importantly, these analyses can be used to find architectural tradeoff points—critical areas of the design with respect to some qualities of interest—and these can become the focus of additional analysis or prototyping as a means of mitigating the risk of building a large, complex software-intensive system.

7. CONCLUSIONS

An ABAS is an extension of the notion of an architectural style. To make architectural styles more rigorous, we associate analytic models of quality attributes with them, in much the same way that Allen and Garlan associate formal semantics with architectural elements to better describe the correctness of a design (Allen and Garlan, 1994). So, an ABAS has associated with it a set of analytic models (such as performance or reliability models) that allow a designer to predict its behavior with respect to some desired quality attributes. ABASs provide to the designer a pre-analyzed structural framework, an analysis, and a mapping between the structure and the analysis.

Associated with the mapping from architectural style to analytic model are two related processes:

1. from a design perspective, there are a set of decisions that accompany turning a style into an implementable design. For example, when decomposing a system's functionality into a set of processes, there is an allocation of functionality to each process, and an allocation of processes to processors. For a performance style we might also make decisions such as choosing the priorities of the processes.
2. from an analysis perspective, there are a set of questions that accompany an architectural style that aid in understanding the style. These questions will ask about the allocation, for example, of processes to processors, their communication mechanisms, the speeds of their connections.

If these questions relate to designs that are repeated over and over again within an organization, then they are often organized into checklists (Maranzano) that are employed during architectural reviews. The answers to the questions form the input to the attribute models. This is the key linkage that comprises the reasoning behind an ABAS: architectural parameters—the things that you can change when you do architectural design—are explicitly related to parameters in an analytic model. In solving the model, we are then modeling the expected behavior of the architecture. The results of this model solving can then be compare back to the expected behavior.

We envision, and are actively working on, a handbook with many ABASs that can be looked to for pre-packaged design and/or analysis wisdom. This is the start of an attempt to make architectural design more of an engineering discipline; one where design decisions are made upon the basis of known properties and well-understood analyses, rather than the currently popular practice of "patch-and-pray".

REFERENCES

R. Allen, D. Garlan, "Formalizing Architectural Connection", *Proceedings of ICSE 16*, (Sorrento, Italy), May 1994, 71-80.

L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Addison Wesley, 1998.

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture*, Wiley, 1996.

E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns—Microarchitectures for Reusable Object-Oriented Software*, Addison-Wesley, 1994.

R. Kazman, G. Abowd, L. Bass, P. Clements, Scenario-Based Analysis of Software Architectures, IEEE Software, November 1996.

R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, "The Architecture Tradeoff Analysis Method", *Proceedings of ICECCS '98*, (Monterey, CA), August 1998, to appear.

M. Klein, T. Ralya, B. Pollak, R. Obenza, M. Gonzales Harbour, *A Practitioner's Handbook for Real-Time Analysis*, Kluwer Academic, 1993.

J. Maranzano, *Best Current Practices: Software Architecture Validation*, AT&T, 1993.

J. McCall, "Quality Factors", *Encyclopedia of Software Engineering* (Marciniak, J., ed.). *Vol. 2*. Wiley, 1994, 958-969.

L. Sha, R. Rajkumar, M. Gagliardi, "A Software Architecture for Dependable and Evolvable Industrial Computing Systems", CMU/SEI-95-TR-005, Pittsburgh, PA: Software Engineering Institute, 1996.

M. Shaw, D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

A Framework for Describing Software Architectures for Reuse[*]

Ezra Kaahwa Mugisa[1] and Tom S. E. Maibaum[2]
[1]Department of Mathematics and Computer Science; University of the West Indies (Mona); Kingston 7, Jamaica; phone/fax : +876 977 1810; e-mail: ekmugisa@uwimona.edu.jm:
[2]Department of Computing; Imperial College of Science, Technology and Medicine; 180 Queen's Gate, London SW7 2BZ, UK; phone : +44 171 594 8274; fax : +44 171 581 8024; e-mail : tsem@doc.ic.ac.uk

Key words: Systematic software reuse, software architectures, object calculus, category theory, component, interconnection, objects, patterns

Abstract: We present a formal description of software architectures for software reuse to support a view of systematic software reuse as the plugging of components into an architecture. The components are object descriptions in the object calculus. Interconnection between the components is defined via synchronisation morphisms within a framework based on category theory. Component composition is defined via the pushout construction, giving the architecture as a "calculated" component, from which the architecture's properties may be derived. We show that the architectures described are reusable in our Reuse Triplet that forms the motivation for our on-going work on systematic software reuse. This work provides further support for the suggestion that category theory provides the appropriate level of mathematical abstraction to describe software architectures.

[*] The work in this paper was supported by various grants from EPSRC and was partially sponsored by the project ARTS (formal Approach to Real-Time Systems development) being carried out by The Laboratory of Formal Methods, Department of Informatics, Pontificia Universidade Catolica do Rio de Janeiro, Brazil, and its research associates, under contract to Equitel SA Equipamentos e Sistemas de Telecomunicacoes. The first author, while on study leave at Imperial College, London, was supported by the University of the West Indies (Mona), Kingston, Jamaica, under the UWI/IDB staff development programme.

1. INTRODUCTION

This work is motivated by a view of reuse-in-the-large that emphasises the reuse of software architectures. The importance of high-level abstraction to the success of reuse has been highlighted in the literature (Krueger, 1992; Biggerstaff and Richter, 1987). That the highest payoffs are to be expected from reusing high-level artefacts such as architectures has been well articulated by some authors e.g., (Krueger, 1992). The systematic reuse of analysis and design knowledge (encapsulated in software architectures) with potentially very high payoffs could help move reuse practice up towards the highest levels on a reuse maturity model.

In order to make reuse-in-the-large a reality, however, we need to have suitable ways of representing and reusing these large-grain software artefacts. Efforts have been made to find suitable ways of representing software architectures for reuse (Terry, et al., 1994; Gamma, et al., 1995; Tracz, 1995). However, one important problem still remains, namely how to find a good formal basis for component composition and interconnection in software architectures.

There are a number of formalisms in the literature for describing software architectures e.g., Darwin (Magee, et al., 1993) and Wright (Allen and Garlan, 1995). In the Darwin model a software architecture is described by a collection of Darwin components, each of which provides services to or requests services from its environment. Darwin components interact by having their service requests connected to appropriate service provisions. This is done by binding their corresponding ports. To instantiate a Darwin architecture, one simply instantiates its components.

In the Wright model an architecture is a collection of computational components together with a collection of connectors, that describe the interactions between the components. The Wright model differs from the Darwin model in that in the former, a connector is an explicit semantic entity. To instantiate a Wright architecture one instantiates the components and the connectors.

Fiadeiro and Maibaum (Fiadeiro and Maibaum, 1996) suggest that category theory provides the right level of mathematical abstraction to describe software architectures. Indeed they show that the category theory approach subsumes the Wright model of architectural description. It could be shown too that much of the Darwin model is similarly subsumed. We thus have a level of abstraction that appears to subsume other known architectural models, and appears to be suitable for performing formal analyses of software architectures in. Here we limit ourselves to issues of reusability.

We view systematic software reuse (SSR) as the process of identifying an appropriate reuse software architecture (RSA) and reuse software components (RSCs) and plugging the latter into the former. The RSA is a template with slots into which RSCs may be plugged. The template may be viewed as an abstraction of a family of systems with slots to be appropriately filled in for each specific system. We may express this view of reuse as the expression SSR = RSA ⊕ RSCs relating the Reuse Triplet (RSA, plugging, RSCs) or diagrammatically as in *Figure 1*. The plugging operator (⊕) takes a collection of reuse components and plugs them into the reuse architecture. The relationship between the RSA, RSCs and the target systems determines whether we are emphasising the reuse of RSAs or RSCs. In (Mugisa, 1997) we apply this view of reuse to well-known examples of reuse.

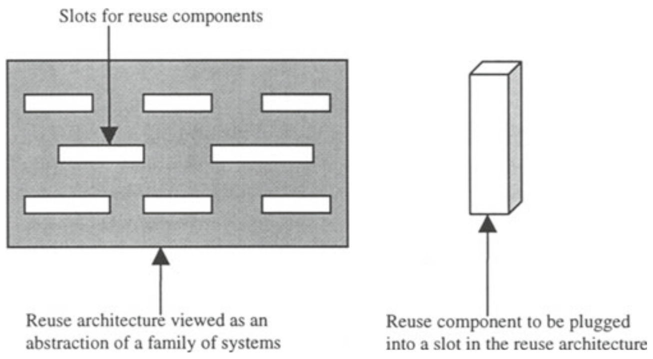

Slots for reuse components
Reuse architecture viewed as an abstraction of a family of systems
Reuse component to be plugged into a slot in the reuse architecture

Figure 1. A view of reuse: SSR = RSA ⊕ RSCs

In this paper we present a framework for a formal description of RSAs and RSCs. We have used the framework to describe the pattern-oriented software architectures of (Buschmann, et al., 1996). Here we will have space enough for presenting only one (simple) architecture. However, we have treated plugging and many of the more sophisticated examples discussed in the literature (Mugisa, 1998). We re-package the architectural patterns of (Buschmann, et al., 1996) at a level of abstraction that is consistent with our Reuse Triplet view. These architectural patterns interest us because the concept of a software pattern is deeply rooted in software reuse. After all a pattern recurs in several different applications from which it is abstracted. In describing architectural design patterns we are describing software architectures that have a high level of reusability.

One simple popular definition of a pattern is “a solution to a problem in a context.” In (Gamma, et al., 1995) design patterns have been presented as “descriptions of communicating objects and classes that are customised to solve a general design problem in a particular context.” Alexander, who initiated the pattern concept, has this to say about the patterns that he used to describe architectures of buildings (Alexander, et al., 1977) “Each pattern describes a problem that occurs over and over again in our environment, and then describes the core solution to that problem, in such a way that you can use this solution a million times over, without ever doing the same thing twice.” Alexander’s concept of a pattern is what we are interested in for systematic software reuse. The other two views tell us how to express patterns.

There are three parts to a pattern : context, problem, solution. We present the context as a domain theory in the object calculus. The problem is a specification in this domain theory; effectively as an extension of the domain theory. The solution takes components of the context domain theory and refines them to an appropriate level of detail. The underlying formal framework is provided by the object calculus : an appropriate temporal logic to express the properties of the basic components and category theory for interconnecting them and synchronising behaviours.

As an illustrative example, we apply our framework to the pipeline pattern from (Buschmann, et al., 1996). This and other architectures also appear in Mary Shaw’s “popular architectural styles” (Shaw, 1995) and in Shaw and Garlan’s “an emerging taxonomy of architectural styles” (Shaw and Garlan, 1996). It is difficult to present a more difficult example because of limitations of space, but the example used is examined in various versions, with the formalism helping to pinpoint the architectural differences and their resulting consequences/properties.

We would like to describe architectural styles in a way that enables us to reason about them so that we can determine interesting properties about them. Our motivation in doing this is well expressed by Mary Shaw (Shaw, 1995b) : “…although many design idioms are available, they are not clearly described or distinguished, and the consequences of choosing a style are not well understood.”

Each software architecture will consist of roles for processing components and in some cases connecting components (connectors) to interconnect the processing components. We shall see how the type of processing and connecting components used affects the resulting architectural style. In all cases the interconnection between the components (either between processing components or between a processing component and a connector) will be described within a category of these components as suggested in (Fiadeiro and Maibaum, 1996). The instantiation of roles by reusable components is also expressed via morphisms in the underlying category (Fiadeiro and Lopez, 1997). A role is essentially a place holder for a processing component as seen from the connector. This is the view from a Wright connector (Allen and Garlan, 1995), for example.

We present each component as a theory description in the object calculus. Each of these theories is encapsulated as an object in the sense of (Fiadeiro and Maibaum, 1992). Each component then becomes an object description and at the same time a theory presentation following the spirit (if not the style) of (Fiadeiro and Maibaum, 1991, 1996).

2. INTERCONNECTING COMPONENTS

We present our architectures as interconnections of computational and connecting components. These components are viewed as theory descriptions represented as objects in Fiadeiro and Maibaum's object calculus (Fiadeiro and Maibaum, 1992). The interconnections are presented as diagrams in a category of these theory descriptions. Let us begin with some basic definitions and propositions.

DEFINITION 2.1 : *a-comp*

The components that are the basic building blocks of our architectures are object descriptions as defined in (Fiadeiro and Maibaum, 1992) and we call each of them an *a-comp* (*a*bstract *comp*onent). An a-comp is a (• , F) pair, where • is the component signature and F are the axioms of the component description.

A typical a-comp has the structure given in *Figure 2*, but there are variations. • = (<data_types>, <action_list>, <attribute_list>) and F = (<axioms>). An a-comp is treated as an object.

```
Component a_comp_name
  data types <data_types>
   actions <action_list>
   attributes <attribute_list>
   Axioms <axioms>
End
```

Figure 2. Structure of an a-comp specification

DEFINITION 2.2 : *sub-object*

Given two objects $\mathbf{obj}_1 = (\bullet_1, F_1)$ and $\mathbf{obj}_2 = (\bullet_2, F_2)$, $\mathbf{obj}_1$ is a sub-object of $\mathbf{obj}_2$ iff the behaviour described by object $\mathbf{obj}_2$ is an extension of the behaviour described by object $\mathbf{obj}_1$, in the sense that the principle of substitutability holds between them. The behaviour of an a-comp (which is an object) is constrained by the axioms F of the $(\bullet, F)$ pair. "The principle of substitutability says that if we have two classes, A and B, such that class B is a subclass of class A (perhaps several times removed), it should be possible to substitute instances of class B for instances of class A in *any situation* with *no observable effect*" (Bud, 1997). The term *subtype* is also used to describe this relationship. This sub-object relation is a form of inheritance for extension, possibly after renaming. It is the inverse of the sub-class relation. This definition should make subsequent discussion of synchronisation based on sharing a common sub-object more intuitive and less confusing. We have the following:

- the signature of $\mathbf{obj}_2$ $(\bullet_2)$ extends a signature isomorphic to the signature of $\mathbf{obj}_1$ $(\bullet_1)$
- F_2 is an extension of F_1, taking into account any renaming that may have been introduced
- $\mathbf{obj}_2$ is a sub-type of $\mathbf{obj}_1$, possibly after renaming

These concepts are expressed through the morphisms of the underlying category.

PROPOSITION 2.3 : *Category a-COMP*

The *a-comp*s and *sub-object* morphisms between them constitute a category *a-COMP*. A sub-object morphism transforms the source object into a sub-object of the target object.

- sub-object morphisms compose, as does inheritance

- there is an identity sub-object morphism - each *a-comp* is a sub-object of itself
- the sub-object relation is associative, as is inheritance for extension

The morphisms may use renaming and so the sub-objects may only be so identified after reverse renaming. We model component composition as a pushout construction within category *a-COMP*. In *Figure 3* component COMP3 is the result of synchronising components COMP1 and COMP2 on their common sub-object SUB, through sub-object morphisms f, g, h and k. SUB is shared by both COMP1 and COMP2, which coalesce around it to form COMP3.

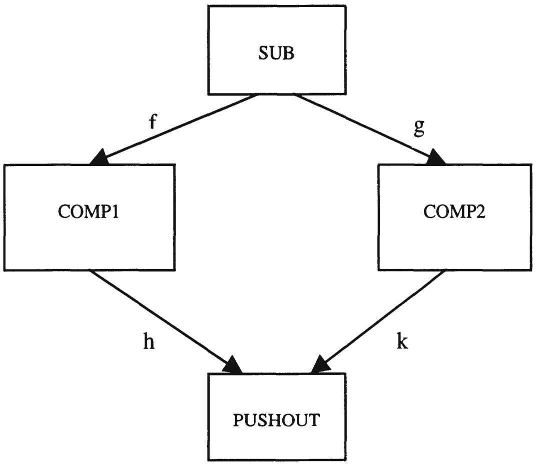


Figure 3. Pushout construction in category a-COMP

DEFINITION 2.4 : *Interconnection*

The style of interconnection that we present here was motivated by the one used in (Fiadeiro and maibaum, 1992). The interconnection of a-comps is governed by the following :

- two components interconnect on a shared sub-object
- the shared sub-object has complementary (or dual) behaviour, e.g a plug and a socket, and it is this duality that makes the interconnection intuitive

This style of interconnection is used elsewhere, e.g., in binding requirements to provisions in the Darwin component abstraction (Magee, et al., 1993). We can view the *Provide* and *Request* ports in a Darwin component interconnection as manifestations of a shared sub-object - a provide/request port.

PROPOSITION 2.5 : *Communication via Ports*

All communication between a component and its environment will be channelled through its communication ports. This is a common model. We use two kinds of ports : ports for message passing (e.g., purely for data transfer) and (possibly implicit) ports for object invocation (see below). A port's *put* and *get* actions transfer data to and from the port's channel, respectively, as defined in Object *Port* below. The port may be viewed as a wrapper around the ubiquitous channel. Channel of T is a channel type capable of transferring data of type T.

```
Object Port
   data types Data, channel of Data
   actions get(Data), put(Data)
   attributes d : Data; ch : channel of Data
   axioms
      get(d) ⇒Xd = ch
      put(d) ⇒Xch = d
 End
```

This port abstraction enables synchronous communication as in CSP (Hoare, 1985) , Occam (INMOS, 1988) and Manna and Pnueli's ("no buffering") channels (Manna and Pnueli, 1992). Asynchronous communication is via a buffer between the communicating components.

DEFINITION 2.6 : *Object Invocation*

Object invocation here follows the CORBA request semantics (OMG, 1996) which states that "when a client issues a request, a method of the target object is called. The input parameters passed by the requester are passed to the method and the output parameters and return value (or exception and its parameters) are passed back to the requester." We use action *request(service-request)* for the action of the source (client) object. The argument (*service-request*) contains the requested service (or method) and parameters. In response, the target object will provide the requested service (if it can) and return results via its arguments. We use action *invoke* for the entire operation covering the request and the response.

Each invocation can in fact be adequately modelled by DMS synchronisation (definition 2.8), with service requests going from DMS-C1's output port to DMS-C2's input port and results going in the opposite direction (see the DMS synchronisation diagram in Figure 5). However, when there are several concurrent service requests, the model becomes messy and the *invoke* abstraction clears away the details. The result is MIS synchronisation as defined elsewhere (Mugisa, 1998).

2.1 We have three sub-objects

We use three types of sub-objects to interconnect the a-comps in our architectures. They correspond to the three ways in which we bind the components. These are Single Port Message-Passing Synchronisation (SMS), Double Port Message-Passing Synchronisation (DMS) and Multi-Port Invocation Synchronisation (MIS). *SMS-Sub, DMS-Sub* and *MIS-Sub* are such minimal sub-objects that can represent a component's ports.

Object *SMS_Sub*
  data types Port
  attributes p : Port
End

Object *DMS_Sub*
  data types Port
  attributes p_1, p_2 : Port
End

SMS-Sub has one port. When used as a synchronising sub-object it takes on both input and output roles in the interconnected components in order to effect message passing from one component to the other. See SMS synchronisation morphisms for details.

DMS-Sub is the 2-port (input/output) version of *SMS-Sub*. It encapsulates two *SMS-Sub* sub-objects.

Object *MIS_Sub*
  data types Service
  attributes $q_1, \ldots, q_n$: Service
End

Sub-object *MIS-Sub* contains services that are mapped to service-requests or to service-provisions. The requests are serviced by the provisions after synchronisation. This sub-object synchronises those components that are linked by object invocation.

2.2 Synchronisation morphisms

The pushout construction synchronises the components on their common sub-object around which they coalesce to form the pushout. There are several structures that the pushout of two a-comps in category a-COMP may have including the structure shown in the component <*pushout_name*> below. Here we discuss SMS and DMS synchronisation morphisms.

DEFINITION 2.7 : *SMS Synchronisation Morphisms*

These define the interconnection of two SMS a-comps by synchronising them on their common SMS sub-object as shown in *Figure 4*. The sub-object morphisms $f_1 : \{p_0 \mapsto out\}$ and $f_2 : \{p_0 \mapsto in\}$ identify *in* and *out* as synchronisation points for the two components, while morphisms g_1 and g_2 are synchronisation morphisms on the sub-objects identified by f_1 and f_2. The synchronisation may be expressed by identifying (or coalescing) the two sub-objects as follows: $f_1 : \{out \mapsto p'\}$ and $f_2 : \{in \mapsto p'\}$. Here is the structure of an SMS a-comp (*SMS_COMP*).

Component *SMS_COMP*
  data types Data, Port
  actions
  attributes *in, out* : Port; *d* : Data
  Axiom
    Relevant behavioural axioms
End

Component *pushout_name*
  Inherit *a_comp1, a_comp2*
  Synchronisation Axioms
    Set of Synchronisation Axioms
End

Figure 4 is the interconnection diagram for SMS a-comps *SMS-C1* and *SMS-C2* yielding pushout *SMS-C3*. (In *SMS-C1*, the *get* action on port *out* is suppressed, i.e., axiom ¬(*SMS.C1.out.get*) holds. Similarly, axiom ¬(*SMS.C2.in.put*) holds for *SMS-C2* to make *in* an input port.) So we actually use a *specialisation* of SMS_Sub in SMS-C1 (and SMS-C2); this is an example of a different form of reuse through inheritance, well known in object-oriented programming and design. This disabling property is maintained by the morphism g_1 (and g_2) as a property of the resulting system. The two components will synchronise on their respective ports, i.e., *SMS-C1.out.put* ≅ *SMS-C2.in.get*. This means that *SMS-C1*'s output action and *SMS-C2*'s input action become synchronised, thus effecting data flow between the two components. This is what interconnecting these two components is supposed to achieve. Synchronisation is expressed, in pushout component SMS_C3, by action *sync* on the relevant ports and by appropriately unifying actions on these ports.

We have introduced the colon notation as an alternative to qualification by the dot notation. We write *A : Exp* to mean that symbols in expression *Exp* are qualified by object *A*. Therefore we may write *SMS_C1 : out.put(d)* as shorthand for *SMS_C1.out.put(SMS_C1.d)*.

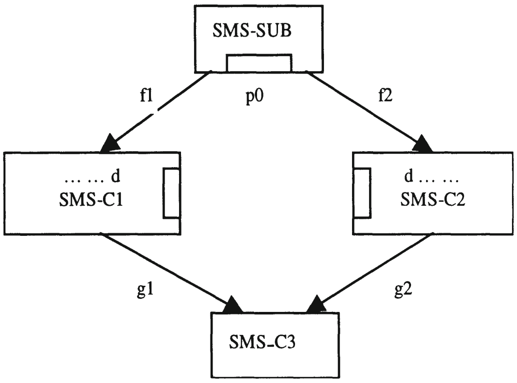


Figure 4. SMS synchronisation diagram

DEFINITION 2.8 : *DMS Synchronisation Morphisms*

These define the interconnection of two DMS a-comps by synchronising them on their common DMS sub-object as shown in *Figure 5*. A DMS a-comp is the double port version of an SMS a-comp. Sub-object morphisms $f_1 : \{p_1 \mapsto out_1 ; p_2 \mapsto in_1\}$ and $f_2 : \{p_1 \mapsto in_2 ; p_2 \mapsto out_2\}$ identify (out_1, in_2) and (in_1, out_2) as pairs of synchronisation points for the two components, while morphisms g_1 and g_2 are once again synchronisation morphisms in the sense

```
Component SMS_C3
  Inherit SMS_C1, SMS_C2
  Synchronisation Axiom
    sync(SMS_C1.out, SMS_C2.in)
    SMS_C1 : out.put(d) ⇔ SMS_C2 : in.get(d)
End
```

described in definition 2.7.

In the diagram of *Figure 5*, a-comps *DMS-C1* and *DMS-C2* are interconnected to yield pushout *DMS-C3*. Action suppression applies as for SMS ports.

The synchronisation diagrams here (in *Figures 4* and *5*) are interpreted as follows:

- morphisms f_1 and f_2 are sub-object morphisms
- morphisms g_1 and g_2 are sub-object and synchronisation morphisms, coalescing the components on the sub-objects identified by f_1 and f_2.
- the pushout (X-C3) (where X is SMS or DMS) inherits both X-C1 and X-C2 and then coalesces them around their common sub-object. It adds synchronisation axioms that translate the synchronisation morphisms (g_1 and g_2) into "equivalence" axioms relating the synchronised attributes and actions.

In both cases the pushout is calculated as an a-comp that extends the inherited theories as discussed above.

Component *DMS_C3*
Inherit *DMS_C1, DMS_C2*
Synchronisation Axiom
sync(DMS_C1.out_1, DMS_C2.in_2)
sync(DMS_C1.in_1, DMS_C2.out_2)
DMS_C1 : $out_1.put(d_1) \Leftrightarrow$ DMS_C2 : $in_2.get(d_1)$
DMS_C1 : $in_1.get(d_2) \Leftrightarrow$ DMS_C2 : $out_2.put(d_2)$
End

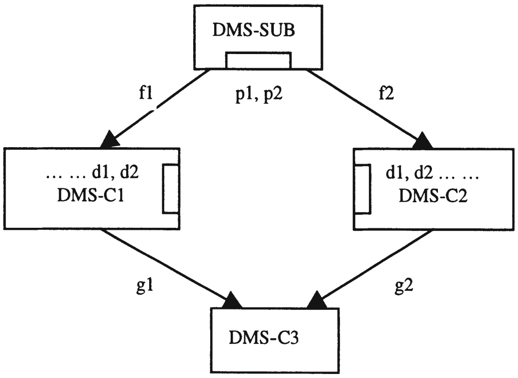


Figure 5. DMS synchronisation diagram

2.3 Asynchronous interconnection of components

In the SMS and DMS Synchronisation diagrams presented above the interconnection between X-C1 and X-C2 is synchronous, requiring the two components to rendezvous in order to communicate. In order to decouple them we interconnect them asynchronously through a buffer, for example as Manna and Pnueli have done in (Manna and Pnueli, 1992). We may use the same synchronisation diagrams to depict asynchronous binding if either X-C1 or X-C2 is a buffered connector. This is how we get the asynchronous connection of *Filter1* to *Filter2* via *Pipe1* in *Figure 6*.

For asynchronous DMS inter-connection we have the equivalent of two pipes going in opposite directions. The details are in (Mugisa, 1998).

2.4 A comparison between synchronous and asynchronous inter-connection of components

It has been stated by C.A.R. Hoare (Hoare, 1978) and others that synchronous communication is the more basic form of communication on top of which asynchronous communication may be implemented as buffered synchronous communication. The trade-off between synchronous and asynchronous inter-connection of components is in decoupling the connected components which must nevertheless separately rendezvous with their buffered connector. We have to depend on the properties of the connector to guarantee that we get the desired asynchronous behaviour from the same components that give us the desired synchronous behaviour.

Manna and Pnueli (Manna and Pnueli, 1992) point out that synchronous communication offers some advantages over the asynchronous version because the execution of a synchronous communication immediately provides the sender with an acknowledgement that the communication has taken place, whereas in the asynchronous case such an acknowledgement has to be explicitly "programmed". The liveness axioms in our buffered connectors give us the same guarantees as the synchronous case except for the delay. On the other hand the asynchronous connection with unbounded buffering gives the decoupled components freedom to exercise independent behaviour without giving up the general properties of the synchronous case except for the introduction of the delay as mentioned above.

3. THE PIPELINE ARCHITECTURE

The "ball of mud" class of architectures is one of those covered in (Buschmann et al., 1996). The ball-of-mud is to be transformed into an organised structure (or a system) by decomposing it into interacting and co-operating sub-systems. The mode of interaction is strongly linked to the chosen architecture. In our setting, a system is defined by an architecture and a set of sub-systems that are instances of more abstract sub-tasks. The ball-of-mud general context is presented below as a domain theory (object *B_O_M_Context*) that states that a system is constructed by plugging a set of sub-tasks into an architecture. The plug action replaces an abstract architectural sub-task by a concrete system sub-task.

```
Object B_O_M_Context
  data types Sub_Task, System, Architecture
  actions plug(Architecture, set of Sub_Task
  attributes arch : Architecture
  Axioms
    $\forall sys : System . \exists subs : set\ of\ Sub\_Task . sys = plug(arch, subs)$
End
```

In this paper we examine only one ball-of-mud architecture known as the pipeline (or pipe-and-filter) architecture. Shaw and Garlan in (Shaw and Garlan, 1996) have this to say about pipes and filters :

"In a pipe-and-filter style each component has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs. This is usually accomplished by applying a local transformation to the input streams and computing incrementally, so that output begins before input is consumed. Hence components are termed *filters*. The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to inputs of another. Hence the connectors are termed *pipes*."

Buschmann et al. in (Buschman et al., 1996) distinguish between active and passive filters and present four pipeline scenarios depending on whether the filters are passive-push (triggered by an active data source), passive-pull (triggered by an active data sink), passive/active-pull/push (triggered by an active filter pulling and pushing) or the more typical all active-pull-compute-push. The passive filters are triggered into a push/pull by direct calls or by data from neighbouring components. This removes the need for pipes and makes the resulting pipelines less interesting for reuse. We shall stick to the more reuse-friendly UNIX-like pipelines of active filters connected by pipes.

In the pipeline architecture the sub-tasks (filters) are arranged sequentially; the output of one sub-task is the input of the next one in the sequence. A pipe component asynchronously connects neighbouring filter components. This system may be specified by the diagram of *Figure 6*. Specifications for all the components of this architecture are presented in the next few sections. The pipeline context given by object *Pipeline_Context* below, simply states that all sub-tasks are filters and that all connectors are pipes.

```
Object Pipeline_Context
  Inherit B_O_M_Context
  data types Filter, Pipe
  Axioms
    Sub_Task ↦ Filter
    Architecture.Connector_type = Pipe
End
```

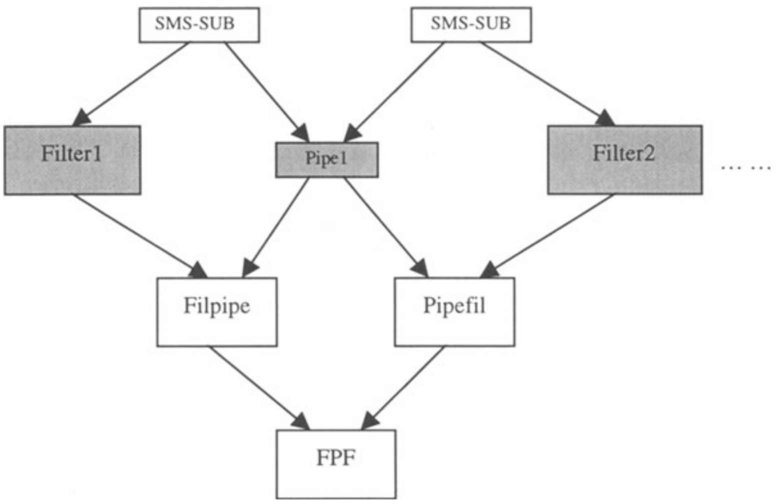


Figure 6. Specification of a system as a pipeline of filters and pipes (asynchronous)

3.1 The components

Component *Filter* below encapsulates our filter a-comp. The liveness axiom guarantees the desired *get-process-put* sequence of actions. Component *Pipe*, the connector, contains a buffer defined as a queue with actions *getq* and *putq*. The safety axioms respectively initialise the buffer, guarantee the absence of an unsolicited response from the buffer and protect buffer update. The second has been included only for emphasis. In our setting it is redundant because it is a consequence of the locality principle (Fiadeiro and Maibaum, 1991, 1992), that states that only the actions declared for an object can change the values of its attributes. For *Pipe*, this may be stated as $(getq \vee putq) \vee ((Xq = q) \wedge (Xin = in) \wedge (Xout = out) \wedge (Xd = d))$. The liveness axiom promises a guaranteed response from the non-empty buffer.

```
Component Filter
  data types  Data, Port
  actions process(Data)
  attributes in, out : Port; d : Data
  Axioms
    $process(d) \Rightarrow \mathbf{X}d = processed(d)$
  liveness
    $in.get(d) \Rightarrow \mathbf{F}(process(d) \wedge \mathbf{XF}out.put(processed(d)))$
End
```

```
Component Pipe
  data types Buffer, Data, Port
  actions getq, putq
  attributes in, out : Port; q : Buffer; d : Data
  Axioms
    $getq \Rightarrow in.get(d) \wedge Xq = q \ @ \ \mathbf{X}d$
    $putq \Rightarrow q \neq [] \wedge q = \mathbf{X}d :: \mathbf{X}q \wedge$
    $\mathbf{X}d = hd(q) \wedge \mathbf{XF}out.put(\mathbf{X}d)\}\}$
  safety
    $\mathbf{beg} \Rightarrow q = []$
    $\neg getq \wedge \neg putq \Rightarrow \mathrm{Xq} = q$
    $\neg(getq \wedge putq)$
  liveness
    $q \neq [] \Rightarrow \mathbf{F}(out.put(first(q)))$
End
```

Instead of a buffered pipe we could have a single item mailbox thus allowing the consumer filter to lag behind its producer filter by at most one data item. This is discussed in section 3.2.2.

Yet another alternative is to have a synchronous pipeline with no pipe connecting the filters. The filters would then be synchronised directly, thus having to rendezvous in order to communicate. Section 3.2.3 has the details.

3.2 Interconnection diagrams

There are three versions of the interconnection diagram corresponding to the three ways of connecting the filters discussed above. We discuss in full the asynchronous version with a buffered pipe and then show how the other two differ from it.

3.2.1 Asynchronous interconnection diagram

To interconnect a *Filter* a-comp with a *Pipe* we use SMS synchronisation of definition 2.7. *Figure 6* is the categorical diagram that shows how two *Filter* components are connected by a *Pipe* in this way.

Components *Filpipe* and *Pipefil* are the local pushouts of the left-hand side and the right-hand side of the interconnection, respectively. They are instantiations of component SMS-C3 specified earlier.

Component *Filpipe*
  data types Filter, Pipe
  attributes *Filter1* : Filter; *Pipe1* : Pipe
  Synchronisation Axioms
    sync(Filter1.out, Pipe1.in)
    Filter1 : out.put(d) ⇔ *Pipe1 : getq*
End

Component FPF is the pushout of the diagram in which *Pipe1* is the common sub-object of *Filpipe* and *Pipefil*. This component is a pushout of two other pushouts and not of simple components. This difference is reflected in the structure of its specification as a component that coalesces two structured components (pushouts) around a connector component as sub-object. The synchronisation axiom reflects this. FPF is also the colimit of the larger (Filter1, Pipe1, Filter2) diagram. All FPF components are similarly connected as suggested in *Figure 6* to define a final colimit (not shown) for the entire diagram.

```
Component FPF
  Instances Filpipe, Pipefil
  Synchronisation Axioms
    $Filpipe.Pipe1 \equiv Pipefil.Pipe1$
End
```

3.2.2 Asynchronous interconnection via a single item mailbox

We take a single item mailbox to be equivalent to a buffer of size 1. We define component *Mailbox_Pipe* similar to component *Pipe* with *buffer* replaced by *mbox*, the single item mailbox type. This connector enables the source filter to lag behind the target filter by only one item, i.e., no new item may be delivered until the previous item has been collected by the target filter.

```
Component Mailbox_Pipe
  data types mbox, Port
  actions getm, putm
  attributes in, out : Port; m : mbox
  Axioms
    $getm \Rightarrow m = [] \wedge in.get(m)$
    $putm \Rightarrow m \neq [] \wedge out.put(m) \wedge \mathbf{X}m = []$
  safety
    $\mathbf{beg} \Rightarrow m = []$
    $\neg getm \wedge \neg putm \Rightarrow \mathbf{X}m = m$
  liveness
    $m \neq [] \Rightarrow \mathbf{F}(out.put(m))$
End
```

3.2.3 Synchronous interconnection

A synchronous interconnection of two filter a-comps is a direct synchronisation of the a-comps. For a transfer of data to take place the two a-comps must rendezvous directly. This would be an instance of SMS synchronisation and is described by *Figure 7*. This is really like composing two functions directly. The tight coupling between the two filter components is evident from the second binding axiom of component FF.

Component *FF*
  data types Filter
  attributes *Filter1, Filter2* : Filter
  Synchronisation Axioms
    sync(Filter1.out, Filter2.in)
    Filter1 : out.put(d) ⇔ *Filter2 : in.get(d)*
End

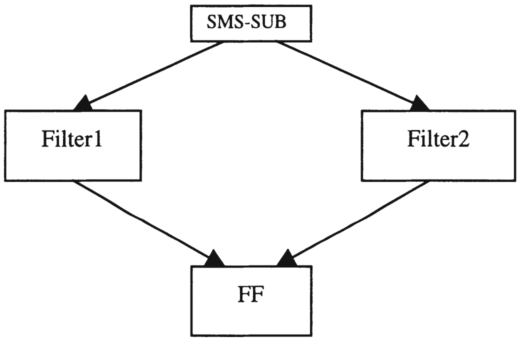


Figure 7. Synchronous pipeline connection diagram

3.3 The pipeline architecture has the pipeline property

In a pipeline connection, if a data item appears at the input port of the first (left) processing component it will eventually appear at the output port of the second (right) processing component. We would like to prove that equation (1) below holds in component FPF. We use the shorthand colon notation introduced earlier as an aid to readability. We get the definitions of *get* and *put* from proposition 2.5. From the definition of *get* we get (1) and from *Filter*'s liveness axiom we get (2). From (1), (2), the synchronisation axioms of *Filpipe* and *getq* for *Pipe*, the value at port *Filter1.in* through attribute *d* and port *Filter1.out* has been added to buffer *Pipe.q*. From *Pipe*'s liveness axiom we shall get (3). From *Pipefil*'s synchronisation axiom and *Filter2*'s liveness axiom we shall eventually get (4) which is what we are required to prove.

$$\text{Filter1 : in.get(d)} \Rightarrow \text{Filter2 : }\mathbf{F}\text{out.put(processed(Filter1:processed(d)))} \tag{1}$$

$$\text{Filter1 : }\mathbf{X}\text{d = in.ch} \tag{2}$$

$$\text{Filter1 : }\mathbf{F}\text{(process(d)} \wedge \mathbf{XF}\text{(out.put(processed(d))))} \tag{3}$$

$$\text{Pipe : }\mathbf{F}\text{out.put(first(q))} \tag{4}$$

$$\text{Filter2 : }\mathbf{F}\text{out.put(processed(Filter1:processed(d)))} \tag{5}$$

Furthermore, it can be proved (using queue operations *getq, putq*) that because we have used a queue to buffer incoming data in the connector, incoming data will be in the order in which it is output by the previous filter in the pipeline.

Since component FPF can be reduced to the structure of *Filter*, if we combine the sequence of *Filter1.process*, the buffer operations and *Filter2.process* into one process, thus also hiding the buffer attribute (and of course its actions) then we get a *Filter*. So we can extend the pipeline to any length we want.

3.4 The architecture

Figure 6 represents the pipeline as an object (the colimit) and as an asynchronous connection of filters. Let us call it component *Pipeline_Arch*. Two other similar figures for the mailbox and synchronous connections can be deduced from *Figure 6*, giving three versions of the architecture. Let us call their representative objects *Pipeline_Arch_Buffered, Pipeline_Arch_Mailbox, Pipeline_Arch_Synchronous*. We therefore have the following expression for the pipeline architecture:

Pipeline_Arch ::= Pipeline_Arch_Buffered | Pipeline_Arch_Mailbox | Pipeline_Arch_Synchronous

3.5 Is the pipeline architecture reusable?

If the pipeline architecture, *Pipeline-Arch*, is an RSA in the Reuse Triplet then it can be reused by plugging in appropriate RSCs. In *Figure 8* RSCs *RealFilter*$_1$ and *RealFilter*$_2$ are plugged into the two *Filter* slots of a pipeline RSA to produce resultant component *RealFPF*. The plugging morphisms are k_1 and k_2. A similar diagram for the synchronous version, can be deduced easily.

The plugging operator must satisfy the requirement that important properties of the RSA are preserved after each slot instantiation. We shall not go further into this topic here, but it has been covered at length in (Mugisa, 1998) under plugging. *Figure 8* appears to contain objects from two categories - the category *a-COMP* and the category of instantiations of a-comps, ie, programs (Fiadeiro and maibaum, 1995, 1997). The existence of a functor between the two categories suggests a way forward.

An alternative way of showing that the pipeline architecture is indeed reusable is to focus on the connector as the central piece in the architecture and to view the components it connects together as its parameters or roles (as the roles of (Allen and Garlan, 1995; Fiadeiro et al., 1997)). We may then show that the connector is reusable by constructing a diagram in which a role (*Filter*) is mapped to its instantiation (*RealFilter*) and completing the diagram with its pushout, component *RealFPF*. See *Figure 8*.

In the plugging diagram of Figure 8, morphisms k_1 and k_2 are the instantiation morphisms of connector *PIPE*'s roles. Morphisms k_{11}, h_1 and k_{12} on the left and k_{21}, h_1' and k_{22} on the right serve to plug the slots in the RSA using the given instantiations. Of course, there is also a role in the connector, ie the buffer. An instantiation of this role by an appropriate buffer implementation would result in an enlarged system (described by a new colimit extending realFPF). Alternatively, one could regard the buffer as an already implemented part of the architecture (hence, describing a less reusable architecture) and represented in the architecture description by the image of the program under the functor that maps programs to their corresponding (minimal, canonic) specifications (Fiadeiro and Maibaum, 1995, 1997).

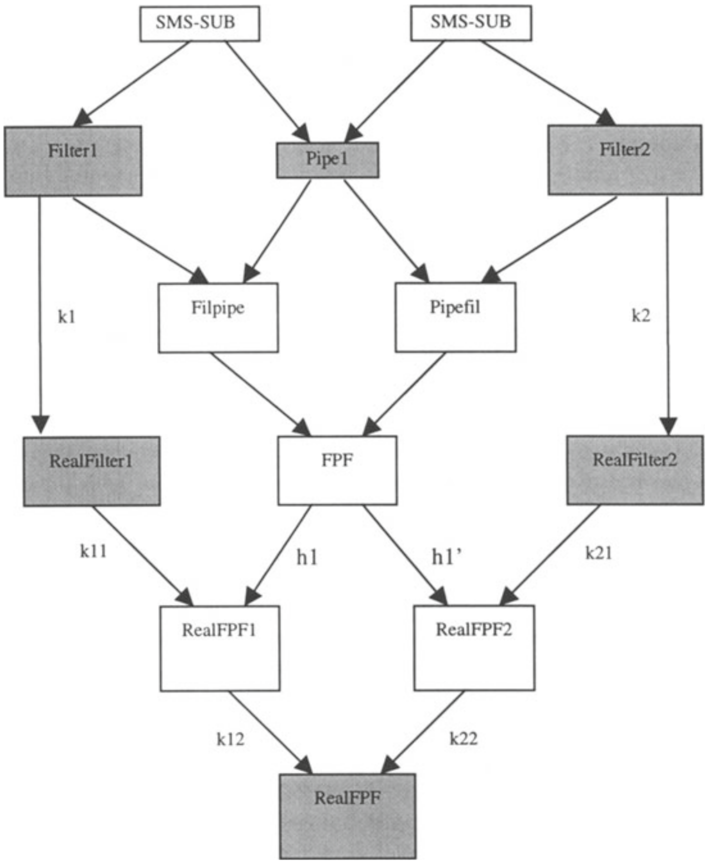


Figure 8. Instantiating a pipeline connection

3.6 Linking context, problem, and solution

We would like to fit *Pipeline_Arch* within the *Pipeline_Context* domain theory and be reassured that all the pieces fit together consistently.

mapping data types : *B_O_M_Context.Sub_Task* $\mapsto$ *Pipeline_Context.Filter*

defining data types : *Pipeline_Context.Filter; ::= Component Filter*

problem : *System ::= plug(subtas*$_1$ .. *subtask*$_n$*, Architecture)* solution : *B_O_M_Context.Architecture ::= Object Pipeline_Arch*

The result of all this is given below in Object Pipeline_Context_Solution and in Figure 9.

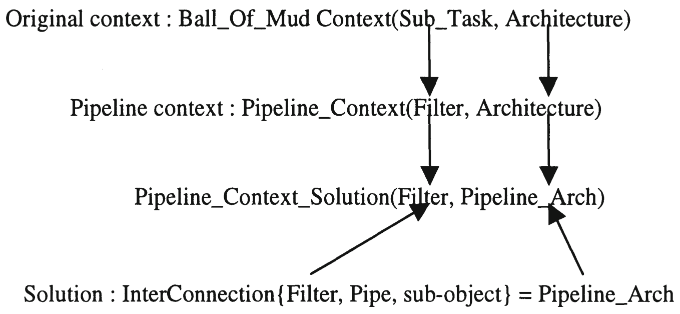

Original context : Ball_Of_Mud Context(Sub_Task, Architecture)
Pipeline context : Pipeline_Context(Filter, Architecture)
Pipeline_Context_Solution(Filter, Pipeline_Arch)
Solution : InterConnection{Filter, Pipe, sub-object} = Pipeline_Arch

Figure 9. Linking context, problem and solution

Object *Pipeline_Context_Solution*
  Inherit *Pipeline_Context*
  data types Filter, Pipeline_Arch
  Axioms
    Pipeline_Context.Filter ↦ Filter
    Architecture ↦ Pipeline_Arch
End

4. RELATED WORK

Shaw and Garlan in (Shaw and Garlan, 1995) discuss the inadequacy of present formalisms to deal adequately with important issues of software architecture. They give examples of Wright and Darwin. We mentioned these formalisms in our introduction. Whereas Wright (with its CSP base) permits static checks such as deadlock freedom as Allen and Garlan did in (Allen and Garlan, 1995), it does not appear to be suitable for issues of dynamic architecture, component composition and interconnection. On the other hand much of the strength of Darwin (with its π-calculus base) is in being able to deal with issues of dynamic configuration of architectures as for example Magee and Kramer have done in (Magee and Kramer, 1996).

The work of Abowd, Allen and Garlan (Abowd et al., 1993) also attempts to formalise and reason about software architecture. We think that the chosen formalism, Z (Spivey, 1992), is not appropriate. The notion of structure in Z (which revolves around the schema) is rather weak and is not suitable for studying what is, after all, a problem of structure. The notion of schema is used more to talk about the textual structure of a specification rather than inherent structure and interconnection as in 'configuring a system out of components'. Also Z does not allow us to talk about behaviours (of components), only about input/output specification of individual operations. To allow us to talk about behaviour, we would need a version of Z that incorporated a temporal logic.

In an effort to find a formalism that adequately deals with interconnection and composition of components for software architecture, Fiadeiro and Maibaum in (Fiadeiro and Maibaum, 1995, 1997) suggested category theory and showed how it subsumed Wright. The work reported in this paper builds on that of Fiadeiro and Maibaum. Our categorical framework also uses as its foundation the work of Joseph Goguen on interacting objects (Goguen, 1991, 1992), especially the principle that "interconnecting systems corresponds to taking colimits in the category of systems, where sharing is indicated by inclusion maps from shared parts into the systems that share them" (Goguen, 1992). Corresponding to Goguen's systems and inclusion maps are what we have called components and sub-object morphisms between them. The general nature of Goguen's categorical framework (as expressed in (Goguen, 1992)) has made our successful application of it to issues of software architecture less surprising than it might have been.

The Kestrel Institute's SpecWare (Srinivas, 1995) is a tool that supports the modular construction of formal specifications and the stepwise and componentwise refinement of such specifications into executable code. Srinivas and McDonald in (Srinivas and McDonald, 1996) report that one of the formal foundations of SpecWare is category theory. They report that the language of category theory has produced for SpecWare a highly parameterised, robust and extensible architecture that can scale to system-level software construction. The colimit operation is their main tool for composing structures, in particular by "gluing" together parts that have overlaps. We use the same operation here to define interconnection of components of an architecture (which are themselves formal specifications) via our sub-object morphism. The operation is the same - "glue" together components along the sub-component that they share.

Rapide's architecture view of "wired interfaces" (Luckham et al., 1995) can be given a categorical semantics through our framework. Since Rapide derives many of its concepts from VHDL (Perry, 1998), we can easily accommodate VHDL's configurations. Rapide's interface and VHDL's entity are abstracted to our component (or slot), simple or structured. Rapide's wired interfaces and VHDL's configurations of connected entities are models of our interconnected components. Mapping entities to their implementing architectures in VHDL's configurations and tying modules to interfaces in Rapide are what we call plugging in our framework. The details of how our framework relates to Rapide and VHDL (two prototyping languages for software and hardware, respectively) are covered elsewhere (Mugisa, 1998).

The work of Moriconi and others (Moriconi et al., 1994, 1995) on architecture refinement is more closely related to plugging in our framework and that is presented elsewhere (Mugisa, 1998). However, architecture composition, which they touch on briefly in (Moriconi and Qian, 1994) is appropriately handled by our framework since an architecture may act as a component of a larger architecture.

5. CONCLUSION

We have presented a framework for describing software architectures for reuse (or RSAs). We present the components of the architectures (or a-comps) as object descriptions in the object calculus. We describe the interconnections between the a-comps using sub-object morphisms between them in a category *a-COMP* of component specifications. An RSA is then derived as the pushout of a categorical diagram that shows how the a-comps are interconnected. This gives us a formal technique for composing (or "calculating") an architecture from its constituent a-comps. We can then derive architectural properties from the resultant a-comps. We have presented an example whereby we used this framework to describe the pipeline architecture and were able to prove one desirable property of this architecture - we called it the pipeline property. We have used the framework to describe other RSAs as well but there is no space in this paper to report on those. In other related work we examine the plugging operator of the Reuse Triplet.

The mathematical underpinnings for this technique were laid out by Fiadeiro and Maibaum in (Fiadeiro and Maibaum, 1995, 1997). In this paper we have applied the mathematics to an engineering problem, namely composing an architecture from (its) components (and with the same structure) as long as we can identify common constituent parts on which to synchronise the components. We can then analyse the resultant architecture using the same tools used on the components it is derived from.

REFERENCES

Abowd, Gregory; Allen, Robert and Garlan David (1993), Using Style to Understand Descriptions of Software Architecture, *ACM SIGSOFT '93 : Foundations of Software Engineering, Software Engineering Notes,* 18(5). ACM Press

Alexander, Christopher; Ishikawa, Sarah; Silverstein Murray; Jacobson Max ; Fiksdahl-King Ingrid and Angel, Shlomo (1977), *A Pattern Language*, Oxford University Press.

Allen Robert and Garlan David (1995), Formalizing Architectural Connection, *First International Workshop on Architectures forSoftware Systems.*

Biggerstaff, Ted and Charles Richter, Charles (March 1987), Reusability Framework, Assessment, and Directions, *IEEE Software.*

Bud, Timothy (1997), *An Introduction to Object-Oriented Programming, Second Edition*, Addison Wesley

Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter and Stal, Michael (1996), *Pattern-Oriented Software Architecture A system of Patterns*, John Wiley & Sons.

Fiadeiro, J and Maibaum, T (1991), Describing, Structuring and Implementing Objects, *LNCS: Foundations of Object-Oriented Languages*, Volume 489, (de Bakker, J. W.; W. P. de Roever, W. P. and Rozenberg, G, editors), Springer-Verlag,

Fiadeiro, J and Maibaum, T (1992), Temporal Theories as Modularisation Units for Concurrent System Specification, *Formal Aspects of Computing*, 4(3).

Fiadeiro, J and Maibaum, T (1997), Categorical Semantics of Parallel Program Design, *Science of Computer Programming*, North Holland.

Fiadeiro, J. L and Lopez, A.(1997), Semantics of Architectural Connectors, *TAPSOFT '97: Theory and Practice of Software Development, LNCS 1214*, (Bidoit, Michel and Dauchet, Max, editors), pages 505 - 519, Springer-Verlag.

Fiadeiro, J. L.; Lopez, A. and Maibaum, T. (1997), Synthesising Interconnections, *IFIP TC2 Working Conference on Algorithmic Languages and Calculi*, (Smith, D. and Finance, J.-P., editors), Chapman and Hall.

Fiadeiro, J and Maibaum, T (1995), Interconnecting Formalisms: Supporting Modularity, Reuse and Incrementality, *Proceedings of the 3rd Symposium on Foundations of Software Engineering*, (Kaiser, G.E., editor), ACM Press.

Fiadeiro, J and Maibaum, T (1996), A Mathematical Toolbox for the Software Architect, *Proceedings of the 8th International Workshop on Software Specification and Design*, IEEE Press.

Gamma,Erich;Helm,Richard;Johnson,Ralph and Vlissides, John (1995),*Design Patterns*,Addison-Wesley.

Goguen, Joseph (1991), A Categorical Manifesto, *Mathematical Structures in Computer Science*, 1(1).

Goguen, J. A. (1992), Sheaf semantics for concurrent interacting objects, *Mathematical Structures in Computer Science*, 2(2), Pages 159 – 191.

Hoare, C.A.R.(1985), *Communicating Sequential Processes*,Prentice Hall.

Hoare, C.A.R.(1978), Communicating Sequential Processes, *Communications of the ACM*,21(8),

INMOS-Ltd (1988), *Occam 2 Reference Manual,* Prentice Hall,

Krueger, Charles (1992), Software Reuse, *ACM Computing Surveys*, 24(2).

Luckham,David C.; Kenney, John J.; Augustin,Larry M.; Vera, James; Doug, Bryan and Mann, Walter (1995), Specification and Analysis of System Architecture Using Rapide,*IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4),

Luckham,David C. and Vera (1995), James, An Event-Based Architecture Definition Language, *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(9).

Magee,Jeff; Dulay, Naranker and Kramer, Jeff (1993),Structuring Parallel and Distributed programs,*IEE Software Engineering Journal*, 8(2).

Magee, Jeff Kramer, Jeff and Sloman, Morris (1989), Constructing Distributed Systems in Conic, *IEEE Transactions on Software Engineering*, 15(6),

Manna, Zohar and Pnueli, Amir (1992*), Temporal Logic of Reactive and Concurrent Systems : Specification*, Springer-Verlag.

Moriconi, Mark and Qian, Xiaolei (1994), Correctness and Composition of Software Architectures, *ACM SIGSOFT '94 : Symposium on Foundations of Software Engineering*, Software Engineering Notes (Wile, David (ed.)).

Moriconi, Mark; Qian, Xiaolei and Riemenschneider, R. A.(1995), Correct Architecture Refinement, *IEEE Transactions on Software Engineering*, 21(4),

Mugisa, Ezra Kaahwa (1997) , A Reuse Triplet for Systematic Software Reuse*, Software Engineering Notes*, 22(2),

Mugisa, Ezra Kaahwa (1998), *An Approach to Systematic Software Reuse Based on Plugging Components into an Architecture*, PhD thesis, University of London, in preparation.

Object Management Group (1996), *The Common Object Request Broker:Architecture and Specification, Revision 2.0.*

Perry, Douglas L.(1998), *VHDL , Third Edition*, McGraw-Hill.

Shaw, Mary (1995), Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging, *Proceedings of the Symposium on Software Reusability (SSR'95)*, (Samadzadeh, Mansur and Zand, Mansour (eds.)), Software Engineering Notes.

Shaw, Mary (1995),Comparing Architectural Design Styles,*IEEE Software*, 12(6).

Shaw, Mary and Garlan, David (1995), Formulations and Formalisms in Software Architecture, Computer Science Today: Recent Trends and Developments. *Lecture Notes in Computer Science 1000* (Jan van Leeuwen (ed.)), Springer-Verlag.

Shaw, Mary and Garlan, David (1996), *Software Architecture: Perspectives on an emerging discipline*, Prentice Hall.

Spivey, J. M.(1992), *The Z Notation : A Reference Manual*, Second Edition, Prentice Hall.

Srinivas, Y. V. and Jullig, Richard (1995), *Specware(TM): Formal Support for Composing Software*, Kestrel Institute Technical Report KES.U.94.5

Srinivas, Yellamraju V. and McDonald, James L. (1996), *The Architecture of Specware, a Formal Software Development System*, Kestrel Institute Technical Report KES.U.96.7

Terry, Allan; Hayes-Roth, Frederick and Erman, Lee (1994), Overview of Teknowledge's Domain-Specific Software Architecture Program, *Software Engineering Notes*, 19(4)

Tracz, Will (1995), Domain Specific Software Architecture, *Software Engineering Notes*, 20(3)

Modeling Software Architecture Using Domain-Specific Patterns

J. P. Riegel, C. Kaesling, and M. Schütze
Dep. of Computer Science, University of Kaiserslautern, Germany,
{riegel, kaesling, schuetze}@informatik.uni-kl.de

Keywords: architectural design patterns, domain-specific modeling support, code generation

Abstract: In this paper we present a domain-specific modeling approach for application components. We use class diagrams and design patterns as major modeling notations and utilize code generation techniques to create an application. Certain architectural aspects of these applications can explicitly be modeled using concrete versions of architectural patterns. As an example, an adaptation of the Pipes and Filters pattern (see Buschmann et al., 1996) is presented, which can be used as an architectural modeling entity and which is supported by a code generator for automatic implementation of different data flow mechanisms.

1. INTRODUCTION

Software components are an important factor in software development. To successfully use a component, its architecture should match to the overall application architecture. This implies that the component architecture must be adaptable with respect to the needs of a specific application. The need for flexibility leads to the questions: "How can the architecture of a component be represented and influenced? Which parts of the software architecture are fixed, which can individually be modeled or varied? Is code generation for architectural aspects possible?"

We try to give an answer to these questions by capturing architectural elements with variants of design patterns and by providing modeling and code generation techniques that allow the user to influence and adapt a components architecture to specific needs. The work presented in this paper is integrated into an experimental, domain-specific development method called PSiGene (*p*attern-based *si*mulator *gene*rator). The goal of PSiGene is to provide a powerful modeling environment to support the creation and integration of customized components. Our initial application domain is building simulation, but support for other domains is possible as well.

In our case, simulators are used in the domain of building automation to test control algorithms. Building simulators must exist in many variants to cope with various physical effects, combinations of effects, required accuracies, and different time advancement schemes (e.g., real-time, time-warp). One complex simulator can not fulfill all possible requirements at the same time, therefore tailored simulation components are required. PSiGene provides a pattern based modeling and code generation environment to support the development of customized building simulators. Section 2 gives a short introduction to PSiGene. For further readings see Schütze et al. (1997) and Heister et al. (1997).

In this paper we present an extension to our initial approach. In order to become more domain independent and to be able to handle more complex models, we emphasized the separation of different component aspects; i.e., we distinguished between component architecture and component behavior. The following figure (*Figure 1*) illustrates the engineering process of PSiGene.

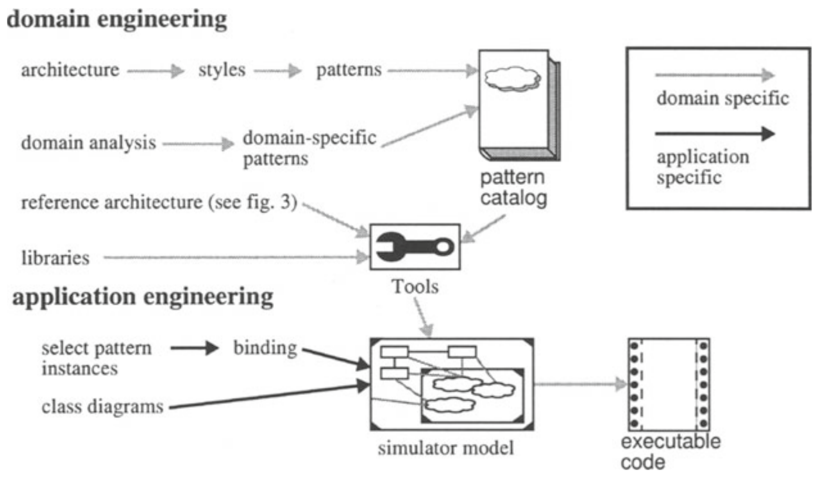


Figure 1. Domain- and application-specific tasks

Some parts of PSiGene, in particular the pattern catalog, the reference architecture, and the libraries are results of a domain engineering step. We tried to capture architectural styles for some component aspects in patterns.

They are designed to work together with domain-specific (behavioral) patterns. All patterns from the catalog form a system of patterns (Buschmann et al. 1996). In addition to the catalog, a reference architecture (see *Figure 3*) was set up, and supporting libraries have been implemented. To design a simulation component (application engineering), appropriate patterns have to be selected from the catalog, instantiated, and bound to class diagrams. Executable code is automatically generated for this application model and can be extended with manually written code if needed.

The following section gives a brief introduction to PSiGene. An analysis of this approach considering software architecture is found in chapter 3. After that, chapter 4 describes two of our architectural patterns (*Pipe* and *Filter*) and gives a short example of their use. A discussion of the approach and an outlook on future work conclude this paper.

2. PSIGENE

PSiGene is a component-based, domain-specific software development approach (for details see Schütze et al. 1997). It's purpose is the creation of tailored, application specific components: in contrast to many component based development methods, where components are provided "as is", PSiGene represents a flexible meta component. The user of PSiGene specifies the concrete component with a model, a generator implements the component automatically from this specification. This results in the creation of components that exactly match the applications needs without introducing any overhead in runtime or memory consumption caused by generic code or interpretation of runtime parameters.

PSiGene combines object-oriented modeling of the static aspects of a component (class diagrams) with pattern-based modeling of the dynamic aspects like component behavior or functionality (pattern instance models), and with code generation techniques for the implementation. The initial application domain of PSiGene is real-time simulation of large buildings.

PSiGene does not work stand-alone, but is integrated into a larger software development environment called MOOSE (*m*odel-based, *o*bject-*o*riented *s*oftware generation *e*nvironment). Within MOOSE, every application consists of a set of components each implementing one aspect of the overall application features. An application is defined by an application model, which in turn consists of several component models. A set of domain-specific generators is used to transform the models into software components. A certain type of generator, the so-called cross-component generator, is capable of interpreting more than one component model at a time and of generating glue logic and application interface code from the interrelations (which we call the "glue") between different component models. PSiGene's generator is implemented as a cross-component generator within MOOSE. More details about MOOSE can be found in Altmeyer et al. (1997).

2.1 System Overview

Figure 2 gives an overview of the implementation of PsiGene. An application, in this case a building simulator, is defined with an application model. Among the different component models we find a structure model (expressed as a class diagram by using editors from MOOSE) that defines the simulation objects. Other class models define structures for other aspects of the simulation or represent run time libraries.

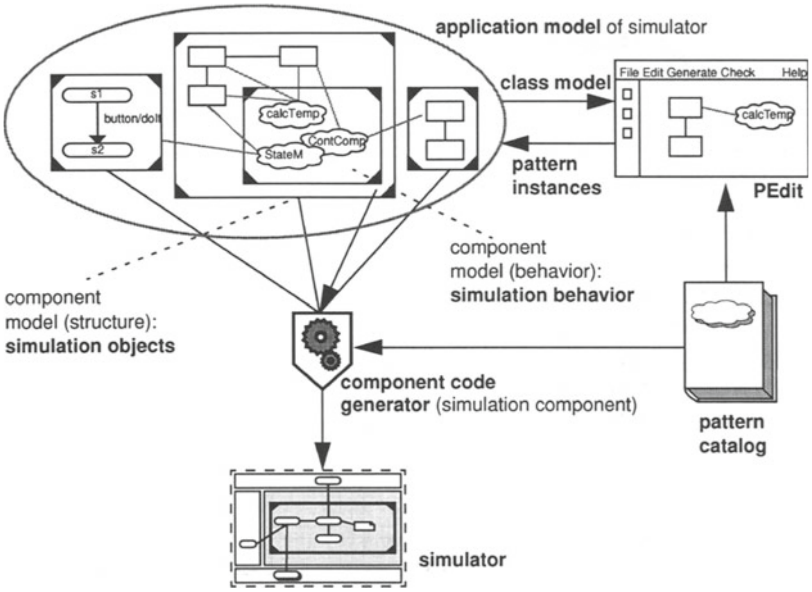

application model of simulator
class model
pattern instances
PEdit
File Edit Generate Check Help
calcTemp
s1
button/dolt
s2
calcTemp
ContComp
StateM
component model (behavior): simulation behavior
component model (structure): simulation objects
component code generator (simulation component)
pattern catalog
simulator

Figure 2. PSiGene overview

The behavior of the simulator is defined with a pattern instance model. Patterns, which are taken from a catalog (see below), are used to define the behavior of the simulation objects, to define the overall functionality of the component, or to define the interface between the simulation component and other components expressed by other models. These pattern instances not only specify local component properties, but also the glue to other components. This means that component integration is also performed on the modeling level. The pattern instances are created using a graphical editor, PEdit, that displays class models and lets the user select and instantiate patterns from a catalog. These instances are then bound to the class model, which means that each instance is connected to elements (classes, relations, attributes, methods) of the class model.

Once the application model is set up, it is fed into PSiGene's generator. The generator reads the structure model, the pattern instances, and it knows the patterns of the catalog. From this information, it creates optimized, tailored component code. For variants of the application, we will simply generate a variant of the component code. Details of pattern-based code generation can be found in Heister et al. (1997).

2.2 Pattern Formalization and Pattern Catalog

As explained before, patterns used for modeling are taken from a domain-specific pattern catalog. The intention of the catalog is pretty much the same as with other pattern-based design methods: to capture successful, "good" design and to provide this knowledge to the catalog user by presenting solutions for smaller design problems in a certain design context. One of the first and most famous catalogs has been presented by the "gang of four"; see Gamma et al. (1995). In contrast to this and most other catalogs found in the literature, which address general design problems, we focus on concrete design problems for building simulation such as the calculation of heat flows in buildings or the scheduling of real-time processes.

Table 1 shows the structure of our catalog. It is partitioned into several categories dealing with different (orthogonal) aspects. As an example, some patterns from each category are shown.

Because we set up the catalog for a very narrow application domain, we are able to state the problems as well as the solution very precisely, enabling tool support for modeling as well as code generation. At the same time, we had to formalize the pattern approach with respect to the pattern interface and the code templates provided as problem solution: In contrast to other approaches, we have to specify the binding between the class model (structure model) of the application and the pattern instances formally and unambiguously. And we need code templates that are suitable for code generation.

Within PSiGene's catalog, the pattern interface, defining the structure as well as the participating elements of a pattern, is expressed with *name:type* pairs as formal parameters. The *name* denotes the name of the participating element, the *type* shows which parts of other component models are eligible for binding, e.g., classes, relations, methods, and so on. Based on the formal parameters, the (syntactical) correctness and completeness of pattern bindings can be checked by tools. With that, the formalization builds the syntactical framework for a pattern language, as the cooperation of patterns can be expressed with formal bindings. Furthermore, the code generator gets sufficient information to create component code.

Table 1. Excerpt from the pattern catalog

Category	Sub-Category	Patterns	Description
Framework Structural Adaptation	Primitive	VariableValue	Access an attribute
		BufferedValue	Attribute buffer that is mainly used in conjunction with a Pipe
	Indirection	FollowRelation	Delegation along a relation
		Traversal	Collect connected objects without specifying a path
	Redirection	MethodBranch	Branch if condition is met
	Pipes and Filters	Pipe	Specify data flow
		Filter	Activity when using a Pipe
	Distribution	AttributeProxy	Used for distributed access
Simulation Control	Control	Actuator	Set attributes with events
		ContinuousComputation	Periodic method invocation
	State Machines	StateMachine	Simple finite state machine
		StateMachineActive	State machine using conditions
User Interface	Display	DisplayAttribute DisplaySlider	Display an object's attribute Display attribute as a slider
Domain	Simulation	ThermalMass	Calculate temperatures
		ThermalJunction	Compute heat flows

The code templates are split into smaller fragments. Each fragment consists of code in a given programming language, enriched with macros that denote the variable parts of the code. Currently, we support Smalltalk as the target language, however, provisions have been made to generate code for other object-oriented languages as well. During code generation, the generator collects the fragments, "personalizes" them by replacing the macros, and assembles the resulting code to methods. Macro replacement can be as simple as string exchange or it can mean to replace a macro with other, complex code fragments recursively. The definition of replacement strategies and code fragments is supported by inheritance and by pattern aggregation.

2.2.1 Extensions to PSiGene

Components generated by PSiGene do not work in isolation, but are embedded into a surrounding application with an underlying software architecture determined by the application domain and other forces. For earlier versions of PSiGene, this application architecture was fixed, and consequently, the component architecture was fixed, too. There were architectural aspects that have been addressed by PSiGene, e.g., the degree of multithreading in a simulator or the possibility to create distributed simulators. However, the decision about architectural elements has been made implicitly, while choosing patterns that determined other simulation aspects. For example, by using the Sensor and Actuator pattern to simulate hardware interfaces, the user implicitly enabled distributed simulation and influenced the component's and application's interface. As we started to apply PSiGene to other application domains, we realized that our approach would become more general and the modeling would be significantly easier if we were able to specify the architecture of applications explicitly. The following section will illustrate how we adapted the latest version of PSiGene (in particular the pattern catalog) to capture and model architectural styles, and how we generate code that implements these styles automatically from the models.

3. SOFTWARE ARCHITECTURE WITHIN PSIGENE

The architecture of a software system can be modeled following architectural styles (see Buschmann et al., 1996, and Bass et al., 1998). Styles give concrete hints on how to construct and organize a system. For example, following the Client-Server style leads to a system where several clients communicate with one or more servers. The exact behavior of a specific client or server is independent of the architectural style and must be specified separately. Tracing which style leads to which component structure makes the software more maintainable and understandable.

Usually several styles can be identified in a component's architecture. Each style can be seen as a set of constraints on an architecture. These constraints define a family of architectures that satisfy them (Bass et al., 1998, p. 25). Some of these constraints can also be expressed with design patterns (compare Monroe et al., 1997). Such a pattern includes the context in which a style can be applied, the forces it resolves, the consequences, and the structure of the style. In addition to this, patterns contain a guide on how to apply them.

Finding concrete patterns that reflect an architectural style is not easy: styles are an abstract description of facets of a software architecture, whereas (PSiGene-) patterns are usually applied to smaller parts of a component and reflect concrete design decisions rather than organizational structures.

We formalized some architectural patterns so that they can be used within PSiGene. Their binding enforces a certain architecture and code can automatically be generated. The main drawback of this "formal" description of architectural styles is that PSiGene patterns cannot capture the whole bandwidth of possibilities how a style can be implemented: only a limited number of domain-specific implementation strategies can be included in a single pattern because otherwise code generation would be impossible and the binding would become far to complex. This restriction, however, doesn't count as much, because our patterns don't aim to be universally applicable but are only used in one domain. When focusing on one domain, architectural styles occur only in few variants.

As explained before, the previous version of PSiGene used architectural styles mostly implicitly: the patterns concentrated on solving a certain (simulation) problem and therefore they contained behavioral aspects as well as structure and other architectural components. For small models this was convenient, but when modeling complex simulators or when adapting PSiGene to other domains it is desirable to be able to model the architecture more explicitly. To do so, we reengineered some of our patterns and added new ones to reflect certain properties of architectural styles.

3.1 Architecture in PSiGene

All simulation components that are modeled with PSiGene share a common basic architecture. Some parts of this architecture are fixed while other parts can vary from simulator to simulator. *Figure 3* gives an overview.

A set of fixed components builds the framework that houses customized simulation components. The framework is used by inheriting from or delegating requests to framework objects or classes. Three major components are used: a GUI library to display simulated objects and to stimulate the simulator, an I/O library to communicate with other applications and to log simulator runs, and the kernel library which is responsible for scheduling and event-handling. The structure and behavior of the simulation components, however, varies for different simulators in order to match the needs of the applications. Variable aspects are:

- component structure (i.e., class models)
- component functionality and behavior
- non-functional requirements (e.g., timeliness, accuracy)
- component integration: glue code to connect to the framework

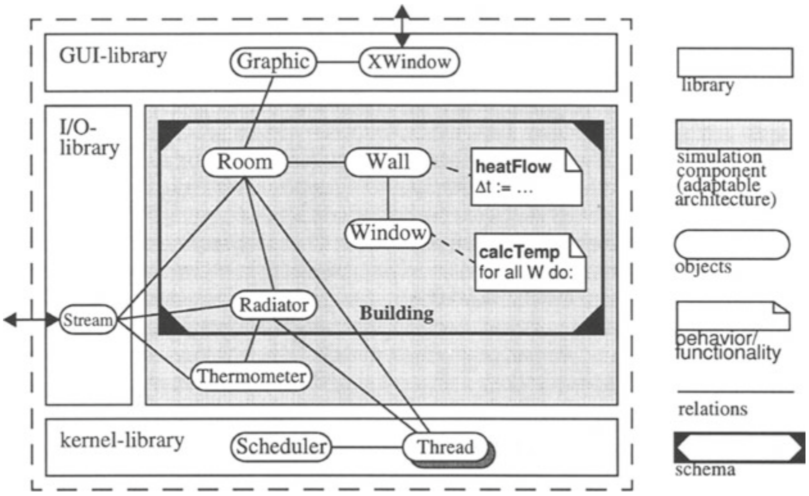


Figure 3. Architecture of a building simulator

3.1.1 Architectural styles in PSiGene

Several architectural styles are used to model a building simulator. The following table (*Table 2*) gives an overview of the styles that occur in PSiGene.

Two styles, Repository and Pipes and Filters describe data aspects of the model. Our components are modeled using class diagrams. Different components can share parts of these diagrams to have access to the same data. Methods to access such a data repository are automatically generated. Data exchange within one component is modeled with the *Pipe* and *Filter* patterns. The communication channels are seen as pipes, and activities to trigger the data flow are modeled as filters (see next chapter).

The application framework implements the framework style. Framework components are represented by class diagrams and can be incorporated into the models using object-orientated mechanisms and patterns.

Since our kernel library is event-driven, all active simulation objects must be able to receive and evaluate events. Event handling is also modeled with patterns (such as *Actuator* or *ContinuousComputation*).

In this section we have shown how the architecture of a simulation component looks like and which architectural styles occur implicitly by using PSiGene. Some styles can also be expressed explicitly with patterns, as we will illustrate with the example in the next section.

Table 2. Architectural styles in PSiGene

Architectural Style	Occurrence	Modeling Notation / Support
Repository	Data exchange between components	Class diagrams
Pipes and Filters	Specify data flow between simulation objects and identify active objects	Pipes and Filters patterns
Framework	Application framework	Class diagrams, schemas, patterns
Layers	Accessing libraries (via delegation)	Indirection, control, and display patterns
Model-View-Controller	Used in the GUI library. The 'Model' is part of the simulation component	GUI patterns
Distribution / Event Systems	Network communication with other applications (I/O library) or distributed simulation/scheduling (kernel library)	Patterns and library parameters
Microkernel	Useful to encapsulate communication aspects esp. in the kernel library	Patterns plus hierarchical class diagrams (not yet supported by PSiGene/MOOSE)

4. EXAMPLE

Up to now, the software architecture of our building simulator models was defined by the framework: the libraries, the structure of our patterns, and by the way the class diagrams are constructed. Many of the patterns addressed behavioral aspects as well as other software architectural aspects.

For example, the *ThermalJunction* Pattern is used to simulate the junction of two adjoining thermal masses. A thermal mass is a simulation object that has a relevant heat capacity. Examples are rooms, radiators, or the environment. A thermal junction is typically a wall or a window. When two thermal masses are adjacent, they exchange energy through heat flows. The *ThermalJunction* pattern can be used to calculate the heat flow between any two of those masses. The heat flow depends on the difference of temperatures of the adjoining thermal masses and on the thermal resistance of the separating (i.e., insulating) material. The first version of the *ThermalJunction* pattern assumes that it can somehow access the required temperatures and the thermal resistance by calling a method. Other patterns like *FollowRelation* or *Traversal* must provide this access methods (usually by delegation to appropriate objects).

Figure 4 shows a part of the model for the simulation of heat flows through a simple wall. The class diagram describes how rooms are connected via surfaces and walls. The lower part of *Figure 4* shows the pattern instances. *ThermalJunction* is bound to the class *Surface* and implements the calculation method. To collect the data for this calculation, several *FollowRelation* patterns are required. The temperature of both neighboring rooms has to be collected and the cumulative thermal resistance of the wall and both surfaces must be computed.

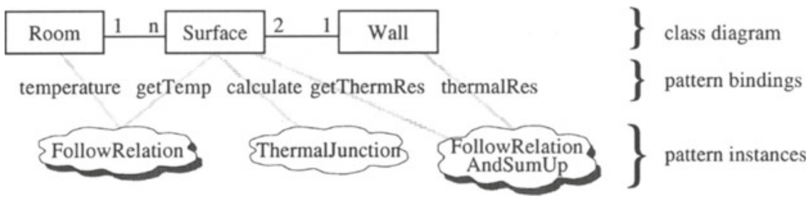


Figure 4. Simulating heat flow

ThermalJunction implements an action (calculation of the heat flow) that is closely related to a data flow (collecting temperatures and thermal resistances). *ThermalJunction* concentrates on the action part and also assumes the required data are present in a certain way. For small object models this is adequate as data flow is relatively simple. As models grow more complex, software architecture becomes more and more important. For the “thermal junction” problem this means, that the data flow aspect becomes more important (and more difficult to model) and the coupling between the data flow and the activity view has to be well considered.

Data exchange between simulation objects usually consists of two parts: a communication channel (object relations or possibly a network connection) and an activity that triggers the exchange. Such pipelines occur in many variants: push-driven, pull-driven, synchronized push/pull, distributed, buffered, and so on. To be able to model such a variety of different data flow possibilities, it is useful to decouple the data flow aspect from the functional aspects and model it separately.

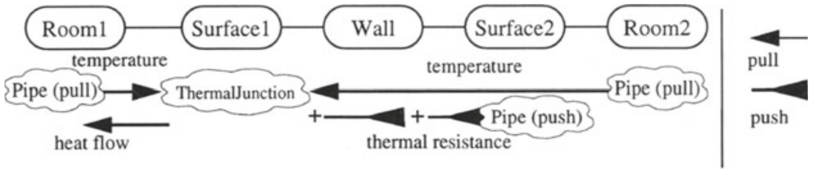


Figure 5. Data flow between two rooms

The new version of *ThermalJunction* focuses on the functional view only. The data to calculate a heat flow must still be present but the pattern doesn't prescribe how to access this data. Two new patterns, *Pipe* and *Filter*, can be used to model data flow. In our example (*Figure 5* and *Figure 6*), we have a data flow (i.e., a pipe) from the class *Room* to *Wall*, and an activity (i.e., a filter) to calculate the heat flow.

Whether the data flow is pull- or push-driven and/or distributed over more processes or computers is characterized by configuring parameters of the *Pipe* pattern. *ThermalJunction* can be seen as a *Filter* (from the data flow view) and bound to our *Filter* pattern (see *Figure 6*).

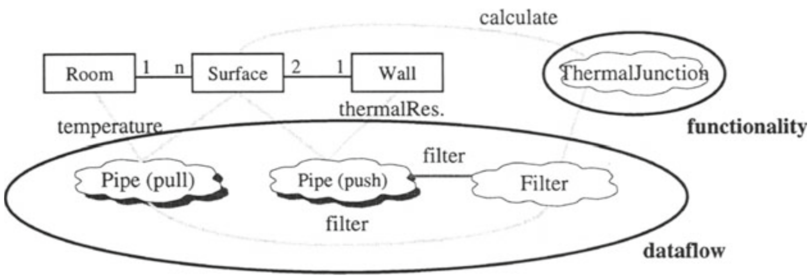


Figure 6. Different views for functionality and data flow

4.1.1 A Pipes and Filters pattern

This section describes our *Pipe* and *Filter* patterns in more detail. It is intended as an example of how software architecture can be expressed with PSiGene-like patterns. We took the pattern Pipes and Filters from Buschmann et al. (1996), which describes most properties of data flows as they occur in our domain (transfer, buffering, synchronization) and adapted it to our needs. The general static structure of a pipeline is shown in the class diagram of *Figure 7*. A pipe is used to connect a provider with one or more consumers. Push or pull methods are used to access data elements in the pipeline. Additional processing is done using filters.

Capturing the idea of the Pipes and Filters style in generative patterns is possible because the underlying structure is not too complex. However, dealing with the many variants in which pipelines occur is not trivial. We have realized the Pipes and Filters style as two individual patterns "*Pipe*" and "*Filter*." They both implement a part of the Pipes and Filters structure (see *Figure 7*).

To identify pipelines in a class diagram, the patterns structure must be mapped to elements from that diagram. This structure mapping is done by assigning values to formal parameters of the *Pipe* and *Filter* patterns (see section 2.2).

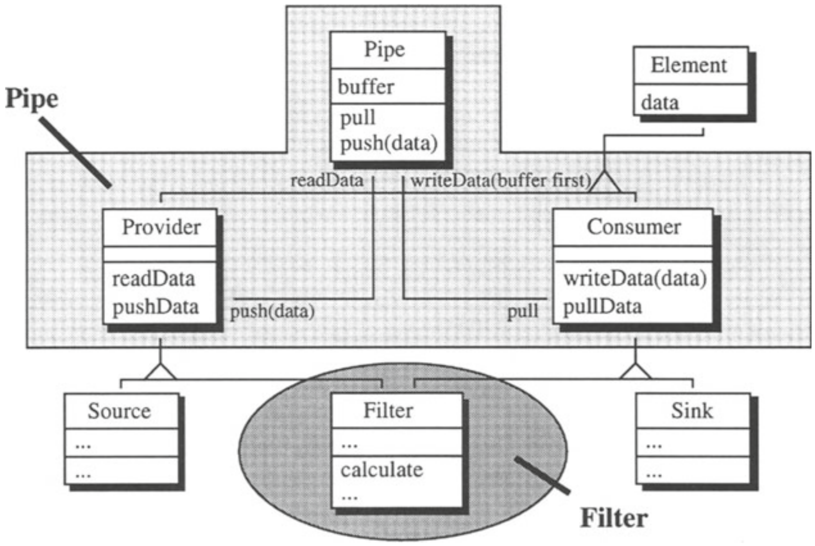


Figure 7. Object structure of the Pipes and Filters style (and pattern)

The following list shows the formal parameters of the *Pipe* pattern:

- **objects**:

  source

  the data source object
  (read as formal parameter *source:object*)

  destination

  the data destination object (sink)

- **attributes**:

  sourceData (use at source)

  the attribute that serves as the source for the data transfer

destinationData (implement at destination)
the new proxy attribute that is the sink of the transfer

- **relations**: entry and exit relations for the pipeline
- **methods**:

  read (optional, implement at destination)
  reads data from the source and writes it to the sink.

  push (optional, implement at source)
  triggers a data transfer. The initiator is the source object.

  pull (optional, implement at source)
  triggers a data transfer. The initiator is the destination object.

  notify (optional, use at destination)
  this method will be invoked at the destination object, if the data at the source has changed.

  request (optional, use at source)
  this method will be called at the data source if the destination object requires an actual value of the attribute. The source object has to transfer the current value (if it has changed since the last time).
- **properties**:

  bufferSize (optional, preset)
  if this property is set, a buffer is realized with the specified size.

  useProxy (optional, preset)
  this property instructs the generators to allow distribution of the participating objects over host-boundaries. A proxy mechanism is implemented.

As one can see, some parameters are optional and don't have to be bound. For example, a push-driven pipe does not need to bind the *"pull"* parameter. The *Filter* pattern is described by similar means. It is bound to calculation methods with a formal parameter *calculate:method(use)*.

Data flow aspects are modeled independently of other aspects. Interaction occurs only at well defined points. Functional patterns like *ThermalJunction* or *ThermalMass* are bound at the filter component using the formal parameter *"calculate."* Activity patterns like *ContinuousComputation* can be bound using the parameters *"pull"* or *"push"* from the pattern *Pipe*, or using the optional filter parameter *"compute."*

A small example demonstrates the pattern bindings for the model in *Figure 6*. After binding values to the formal parameters for all pattern instances, a binding description file is created. It looks as follows:

```
"ThermalJunction 1 - calculates the heat flow through a thermal junction element"
ThermalJunction
   bind: 'target' to: 'Surface';
```

```
    bind: 'calculate' to: 'calculateHeatFlowForRoom';
    bind: 'thermalResistance' to: 'thermalResistance';
    bind: 'area' to: 'area'.

"Filter 1 - calculates the heat flow through a thermal junction element integrating the pattern
    instances ThermalJunction1 and Pipe1 to Pipe5"
Filter
    bind: 'target' to: 'Surface';
    bind: 'calculate' to: 'calculateHeatFlowForRoom';
    bind: 'request' to: 'requestHeatFlowForRoom';
    bind: 'arguments' to: #('temperatureOfRoom' 'temperatureOfSurface' 'resistanceOfRoom'
    'resistanceOfSurface' );
    bind: 'getArguments' to: 'argumentsHeatFlowForRoom';
    bind: 'notify' to: 'notifyAttributeChangedForHeatFlowForRoom';
    bind: 'result' to: 'heatFlowForRoom';
    bind: 'initValue' to: '0.0'.

"Pipe 1 - provides the thermal resistance of a room at a connected surface"
Pipe
    bind: 'source' to: 'Room';
    bind: 'destination' to: 'Surface';
    bind: 'sourceData' to: 'thermalResistance';
    bind: 'read' to: 'readResistanceOfRoom';
    bind: 'exit' to: 'radiatorSurfacesOfRoom';
    bind: 'entry' to: 'roomOfradiatorSurface';
    bind: 'destinationData' to: 'resistanceOfRoom';
    bind: 'push' to: 'pushThermalResistanceToSurfaces';
    bind: 'notify' to: 'notifyAttributeChangedForHeatFlowForRoom'.

"Pipe 5 - provides the calculated heat flow from Surface to Room"
Pipe
    bind: ‘destination’ to: ‘Room’;
    bind: ‘source’ to: ‘Surface’;
    bind: ‘exit’ to: ‘roomOfSurface’;
    bind: ‘entry’ to: ‘surfacesOfRoom’;
    bind: ‘read’ to: ‘readHeatFlowFromSurface’;
    bind: ‘destinationData’ to: ‘heatFlowFromSurface’;
    bind: ‘sourceData’ to: ‘heatFlowForRoom’.
    bind: ‘pull’ to: ‘pullHeatFlowFromSurface’;
    bind: ‘request’ to: ‘requestHeatFlowForRoom’.
...
```

There are 3 objects classes in this example: a *Room* is connected with a *Surface* to a *Wall*. The instances of *Room* have to recalculate their temperatures in fixed time intervals. The rooms request the calculation of the heat flows from the adjoining objects indirectly by reading the local attribute “heatFlowFromSurface” (the calculating filter for the temperature at *Room* calls pullHeatFlowForRoom first, before accessing the attribute).

Since the generators know that rooms and surfaces are connected by a one-to-many relation, this attribute (implemented by "Pipe 5") holds a collection of heat flow values. Each of these values is calculated by an instance of the *ThermalJunction* pattern, the calculation is controlled by "Filter 1". This filter collects all required arguments, triggers the calculation while providing those arguments, and, if necessary, delivers the result via "Pipe 5". The following code fragment was generated from the above bindings:

```
argumentsHeatFlowForRoom
    "Collects all arguments for the calculation of heatFlowForRoom"
    | args |
    args := Array new: 4.
    args at: 1 put: self temperatureFromRoom.
    args at: 2 put: self temperatureFromWall.
    args at: 3 put: self thermalResistanceFromRoom.
    args at: 4 put: self thermalResistanceFromWall.
    ^args

computeHeatFlowForRoom
    "Does the calculation and stores the result in heatFlowForRoom"
    ^self heatFlowForRoom: (self calculateHeatFlowForRoom: self
    argumentsHeatFlowForRoom)
```

In our example the access to "heatFlowFromSurface" is triggered by a separate pull method (Pipe 5 is pull-driven). The other pipes are push-driven, which means that the data transfer is initiated by the sources of the pipe.

As one can see, our patterns "*Pipe*" and "*Filter*" realize a flexible data flow mechanism with synchronization capabilities. They separate this aspect from the functionality, which in this case is handled by *ThermalJunction*. *ThermalJunction* in turn does not care about data flow issues.

Depending on the binding, different transport and synchronization mechanisms can be implemented by *Pipe* and *Filter*. Some synchronization combinations are shown in *Table 3*.

Table 3. Some possible combinations of pipes and filters

Argument Pipes	Filter	Result Pipe	Comment
push-driven	inactive	push-driven	Each time a new argument is delivered, the result is calculated and propagated.
pull-driven	active	push-driven	The calculation of the filter is triggered by an external activity. The arguments are requested and the result is propagated.
pull-driven	inactive	pull-driven	If someone requests the result, it is

Argument Pipes	Filter	Result Pipe	Comment
			calculated after requesting all required arguments.
push-driven	active	pull-driven	This is the synchronized combination of the first three examples

The required synchronization mechanism depends on the frequency of data changes and on how the transport is triggered. The pattern interface allows to abstract from the concrete transport mechanism. Distributing the pattern or buffering values can be achieved by binding additional patterns like *AttributeProxy* or *BufferedVariable*.

4.1.2 Code generation

Each pattern instance in PSiGene comes with a partial code generator. It is responsible for generating adequate code from the patterns code templates and the pattern bindings. Every pattern instance is analyzed in its binding context before the generation is started. Therefore, tailored and optimized code can be created.

To generate code for a pattern, not only its own bindings have to be considered, but also other patterns bound to the same target objects. For example, a propagating filter needs information about the pipe to which data changes should be reported. Internal properties (additional bindings) are used to allow the combination of patterns and are used to generate optimized code. Application code is generated by assembling tailored code templates that are part of each pattern. A very simple code fragment may look as follows:

```
'{compute}
   "Does the calculation and stores the result in {result}."
   ^self {result}: (self {calculate}: self {getArguments})'
```

Keywords in brackets ({}) are used as macros. Usually code generation can be done by choosing code templates and replacing all macros with other templates or bound values. More complex patterns (like *Traversal*) also use code synthesis techniques. For further reading see Heister et al. (1997).

5. DISCUSSION

This work combines different software engineering techniques. Structure models are used together with a pattern based design strategy. Application generators are used to implement a simulation component. The approach can be seen as a domain specific software architecture (DSSA, see Mettala and Graham (eds.), 1992). Domain engineering in the field of building simulation resulted in the overall architecture of a simulation component (*Figure 3*) and in the implementation of the libraries. Also our pattern catalog is domain-specific and part of the domain model. Reference requirements are included in the informal parts of our patterns, prescribing which patterns could be used together or giving hints how certain simulation problems can be solved. To design a simulation component, only the application engineering has to be performed. This includes especially setting up or refining a class diagram for the building structure and instantiating and binding patterns from the catalog. Tool support is given for these tasks. A detailed process of this modeling procedure is not yet defined and will be a topic for future works.

The revised pattern catalog contains behavioral patterns together with patterns describing architectural styles. It is partitioned into categories that deal with different aspects of simulation (the partitioning supports aspect oriented programming (AOP), see Kiczales, 1997). Each category can be seen as a view and be modeled separately. We are currently extending the pattern editor to support views.

The main advantage of the new catalog is that we have found a way to express parts of the component's architecture in (design) models. Patterns can be used to implement or refine an architectural style. The configuration is done by binding a pattern instance to the simulator model. These patterns have a fixed formal interface and code templates (see Heister et al., 1997) and therefore cannot express the whole variety of a style. But as our domain is limited, it is sufficient to use only a few domain-specific implementations of an abstract style.

All our patterns must be able to work together: they form a system of patterns (compare Buschmann et al., 1996). Each individual pattern is used to model a part of the component, but with the right combination of patterns a building simulator can be designed. For example, with our *Pipe* and *Filter* patterns, a data flow between two objects can be defined. At both ends of the pipeline activity may take place. This is usually a calculation of values. The Filter pattern is used to trigger such an activity; the activity itself must be modeled elsewhere (i.e., with patterns from another category). A future topic is to investigate how such dependencies and constraints between patterns can be formally expressed.

6. CONCLUSION

With architectural patterns it is possible to model architectural styles separately. The pattern binding concept of PSiGene allows to combine different pattern instances and to apply them to class diagrams. Therefore, architectural styles can be integrated using a formal interface. The main advantage of the architectural patterns is a better maintainable system (model changes take effect only locally), better tailored components, and the ability to handle more complex models. Also, our pattern catalog became more domain independent. We still have some special simulation patterns but they are able to work together with more abstract and more general architectural patterns. We believe that these architectural patterns can easily be adapted to other domains. Future work will investigate the applicability of our approach in other domains.

Our patterns do not provide the whole bandwidth of all of their possible applications but only a domain-specific subset. This makes code generation and optimization possible but restricts the universal usage of the patterns a bit. Finding more variants and new patterns is also a topic for future works.

A small disadvantage of our new pattern catalog is that it takes more time to model small simulation components as each aspect has to be designed separately. But for larger models this separation of concerns is mandatory and leads to more flexible simulators (e.g., nonfunctional requirements like distribution can be modeled and documented explicitly and more easily).

The pattern-based approach to software architecture seems to be feasible and worked well for PSiGene. Variants of components can be created within short time, and a component can match the architectural demands of an application by changing abstract architectural properties in the models. We therefore believe that our approach to architecture modeling helps the software development in providing and using tailored components.

REFERENCES

Altmeyer, J., Riegel, J. P., Schürmann, B., Schütze, M., and Zimmermann, G. (1997) Application of a Generator-Based Software Development Method Supporting Model Reuse, *9th Conference on Advanced Information Systems Engineering (CAiSE)*, Barcelona

Bass, L., Clements, P., Kazman, R. (1998) Software Architecture in Practice, *SEI series in software architecture*, Addison-Wesley

Batory, D., Singhal, V., Thomas, J., Dasari, S., Geraci, B., Sirkin, M. (1994) The GenVoca Model of Software-System Generators, *IEEE Software*, September 94

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stall, M. (1996) *Pattern-oriented Software Architecture - A system of Patterns*. John Wiley & Sons Ltd.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995) *Design Patterns*, Addison-Wesley

Heister, F., Riegel, J. P., Schütze, M., Schulz, S., and Zimmermann, G. (1997) Pattern-Based Code Generation for Well-Defined Application Domains, *European Pattern Languages of Programming Conference (EuroPLoP)*, Siemens Technical Report 120/SW1/FB, 263-273

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. Irwin, J. (1997) Aspect-Oriented Programming, *PARC Technical Report*, February 1997, SLP97-008 P9710042

Kim, J. J., Benner, K. M. (1996) An Experience Using Design Patterns: Lessons Learned and Tool Support, *Theory and Practice of Object Systems*, Vol. 2(1), 61-74

Kruchten, P. B. (1995) The 4+1 View Model of Architecture. *IEEE Software*, 42-50, November 1995

Lieberherr, K. J. (1996) *Adaptive Object-Oriented Software Development: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston

Mettala, E., Graham, M. H., eds. (1992) The Domain-Specific Software Architecture Program, *Special Report CMU/SEI-92-SR-9*, Carnegie Mellon University, Pittsburgh

Monroe, R. T., Kompanek, A., Meltom, R., Garlan, D. (1997) Architectural Styles, Design Patterns, and Objets, *IEEE Software*, January 1997

Schütze, M., Riegel, J. P., and Zimmermann, G. (1997) A Pattern-Based Application Generator for Building Simulation, *European Software Engineering Conference (ESEC)*, Zürich

ImageBuilder Software
A Framework Development Experience Report

Dwayne Towell
ImageBuilder Software
6650 SW Redwood Lane, Suite 200
Portland, OR 97224
dwayne@imagebuilder.com

Key words: Framework, experience, object-oriented, cross-platform, multimedia, reuse, education, domain, team, architects, development

Abstract: Six years ago ImageBuilder Software chose to develop an object-oriented, cross-platform, multimedia framework to promote code reuse and therefore increase profits. Today, it continues to be used and extended; it is profitable and a major company asset. This paper documents how it was developed, describes how we use it today, evaluates its success, and makes recommendations for others based on our experience.

1. THE COMPANY

ImageBuilder is an independent, full-service, multimedia title development company. Founded 15 years ago, it now has over 120 full-time employees including more than 30 engineers. We design, develop and test CD-ROM titles for clients, partners and, more recently, our subsidiary Active Arts. Many products are completed independently from conception through development to manufacturing release by ImageBuilder staff, however we do allow clients to participate in the process to the extent they desire. Most of the dozen or so products shipped each year are dual Windows and Macintosh, shrink-wrap, edutainment, multimedia CD-ROMs. Some of the most well-known titles include: Hasbro's *Pictionary, Mr. PotatoHead*, and *Playskool Puzzles*; Creative Wonder's *Madeline Classroom Companion* series; The Learning Company's *Math Munchers*

Deluxe and *Paint, Write, and Play*; Pacific Interactive's *Bill Nye: Stop the Rock!*; Disney's *Disney Magic Artist*; and Microsoft's *Arthur's Playground.*

2. PROJECTS

Each project is somewhat unique, however most follow a well-traveled path. ImageBuilder producers collaborate with the client to develop an outline for the product including scope, content and purpose. As this is nearing completion, the project lead and art director are assigned to begin developing a product specification. This is a working document so changes are quite regular especially near the start of project as details are ironed out with the client, engineers and artists.

As areas of the product specification become firm, engineers start on the technical specification. Engineers outside the project will usually critique it in one or more design review meetings. Once the product specification is complete, the Quality Assurance department will develop plans for testing and certification. Additional members are added to the project as they are needed and/or become available. Eventually, the team will include two to six engineers, one or more media coordinators, artists, animators, scriptwriters and/or sound designers. Most projects employ about eight full-time positions and tend to last about ten months, but they can vary quite a bit. At any time, ImageBuilder has a dozen or more projects in progress.

3. SUPPORT

One important ingredient contributing to a successful project at ImageBuilder is our proprietary, object-oriented, multimedia framework and its associated tools. Framework code comprises from one third to one half of most applications developed. Typical applications make use of about 80% of the code provided. Since practically all development builds on our framework, a framework development team continues to improve it and provide support for its use. The framework team's initial responsibility is to develop, improve and extend our domain-specific object model for multimedia applications. This model is realized as an object-oriented C++ framework. Several tools have also been developed to allow the large quantities of multimedia resources to be manipulated, compiled, and viewed.

In addition to developing the object model, delivering code and providing tools, the team provides design review and education. Most projects take advantage of the team's design experience during formal reviews. Also, engineers frequently use the team members as a convenient consultant for object design. Education provided by the team plays an important part in the improvement of the engineering department. Whenever consulted the team tries to take advantage of these "teachable" moments. In addition, formal classes are held weekly covering various topics from engineering processes to use of the framework to object modeling.

4. ARTIFACTS

The framework team maintains three artifacts: C++ framework, documentation and tools. The framework is actually multi-tiered; it includes an operating system services (OS) layer, an application layer and the multimedia framework proper. The OS layer models services such as files and threads, providing cross-platform objects to hide platform-specific details. The application layer builds on the OS layer to provide windows, controls, menus, events, etc., again hiding platform-specific details as much as possible. Finally, the framework proper models multimedia objects such as pictures, sounds, animations and buttons.

Table 1 shows the number of modeling objects and additional utility classes for each layer.

Table 1. Domain Objects by Framework Layer

	Domain Objects	Other Classes	Total
OS services	26	53	79
Application framework	20	27	47
Multimedia framework	88	118	206
Total	134	198	332

In addition to framework code, extensive documentation has been developed. Approximately 60 pages of overview and intermediate-level documentation are currently available on an internal web site. This continues to grow as the framework is extended or new areas are identified for further explanation. Class-level documentation, aimed at application use, is provided with the class in header files.

The framework team also has responsibility for several tools including a content compiler, a script compiler, an animation viewer, a resource browser as well as others. These tools allow resources to be converted, compiled and viewed for shipping and run-time use. Resources supported include: text, pictures, sounds, MIDI, run-time composed and streamed animations, QuickTime movies, Windows AVIs and project-dependent extensions.

5. HISTORY AND EVALUATION

Seven years ago ImageBuilder began to respond to industry changes as Windows 3.0 and 3.1 were shipped. Emphasis changed from business graphics to multimedia products. Six years ago we started our third multimedia project and decided much of each project could be reused if we could developed a "multimedia engine". So a parallel project to create reusable code was started in conjunction with the client's project. Although extremely primitive by today's standard, it gave us our first chance. The most important thing we learned was the need for extension—the need for a framework, not an "engine". It also allowed us to "throw one away".

At the conclusion of that project we immediately went back to the drawing board. We allocated two of our best engineers and completely redesigned the object model from the ground up. Over the next year or two we continued to improve and extend the model. Several projects were now using it and a few had even been completed. By the end of this period we shipped about 6 products using the framework, but had also come to realize many of its shortcomings.

The third year into the project we went back to the drawing board again. Although we eventually touched all the code, much of the domain-related object model remained unchanged. Following this redesign was an extensive period of conversion. The framework team was under pressure to deliver vast amounts of new code for projects underway. In response we tried using occasional part-time team members either to help with conversion or develop new areas. In the end, this did not work because our part-time engineers did not have experience developing extensible code.

As the percentage of projects using the framework increased to almost 100% and the engineering department grew, we had to improve our processes. We added an administrative assistant and an official release procedure. Releases now included detailed change reports, verified code for all supported platforms, and tools synchronized with the run-time code.

Over the years many additions have been suggested for the framework. We have never lacked for proposed improvements and as each project pushes the envelope, pressure to make enhancements increases. The framework team periodically reviews the framework with two groups. Producers are consulted for strategic direction. Engineers are consulted about utility and convenience. Based on these directions the framework team develops short- and medium-term goals.

In addition to application team needs, framework additions must meet two requirements. First, the addition should be useful to more than one project since the framework exists to reduce costs by increasing code reuse.

Second, the addition must be well defined. Open-ended additions become sinkholes for time and effort.

The need for education was one of the slightly unexpected results of developing a framework. Although conversant in the C++ language, many of our engineers lacked object skills. In an attempt to improve skills and deliver timely information about the framework, weekly engineering meetings were established. Time is divided between teaching object skills, improving engineering practices, and discussing framework use.

The framework has been a major benefit to project development at ImageBuilder. It allows application teams to do work faster because many of the details are already solved. For example, rapidly prototyping a game, module or entire application can be done as fast as content becomes available. Also, it improves the quality and speed of applications ImageBuilder is able to ship. Each product that takes advantage of the framework contains proven and optimized code for its core functionality.

6. RECOMMENDATIONS

6.1 Getting Started

Building the framework team is the first step to a successful framework development. The team will be responsible for deriving the object model, documenting it for project engineers, implementing it in code, and supporting it.

6.1.1 Create a Framework Development Team

You will not have well-architected, reusable code if no one has primary responsibility to develop it. As much as it would be nice to believe good engineers would develop reusable objects and code, it is very difficult while under pressure to meet deadlines. Our experience indicates it will never happen. However, it can be accomplished given a team whose full-time responsibility is framework extraction or invention.

6.1.2 Enroll the Best Architects

Design is first chance engineers have to influence a project for success. Correct decisions pay off for the rest of the project, while mistakes made here cost the most to correct. Given a good domain-specific framework, much of the design for a project has already been done. So, to make the most of your investment, allocate the best architects available to framework development. This allows all projects, and engineers, to benefit from their experience and knowledge.

6.1.3 Allocate Ten Percent

Allocate enough engineers to be productive and make a real contribution, but not too many to be unmanageable or risk extensive overhead. Our team has varied slightly over the years, but we have found allocating ten percent of our engineers to be about right.

6.1.4 Promote Stable Membership

Team membership should be stable. Framework development is fundamentally different from application development. For example, correct framework design is typically more important than the schedule. Also, inventing a quality model is considerably different from using it. Therefore, enroll new members for long terms, a year or more, and make key members permanent. Since members will be working together for extended periods, and in some cases indefinitely, chose the team carefully and use trial periods. This allows everyone to reevaluate the assignment after an agreed interval. If the situation does not appear to be working, the trial can be terminated with less discomfort for everyone involved.

6.1.5 Empower a Visionary Leader

As with all enterprises, a strong leader is needed. Someone who has a clear vision for the framework needs to "own" the project. Especially before the project is well established, but even later, it will be pulled in many directions. Each client project will make a case that the framework team should solve its special requirements. An empowered leader will be able to hold it on course and allow it to meet the widest possible needs.

6.2 Making Progress

Once the project gets underway the team will be developing the domain object model. It is unlikely any team will "get it right" the first time around. Even if the first version is successful, plan to improve and extend it.

6.2.1 Make It Tractable

Only consider taking on manageable areas of the domain. Some areas will be complex or ill defined; ignore them. In many cases the application engineers will be able to solve the subset of the problem needed for their application without needing the general solution. Even though a general solution may be enticing, especially to architects, consider the economics of building it.

6.2.2 Make Official Releases

As with any construction project, building on an unstable base is tricky at best. Give the application engineers a hand by producing "official" code releases. Produce "release notes" that announce model changes, extensions and bug fixes. Typically project engineers are much happier to receive new code if they know what has changed. Balance the desire to release changes to application engineers with the cost to perform a release. We make releases at most once a week, but they may occur less frequently if few changes have occurred.

Suggest each project archive its own copy of the framework. This allows an application member to control migration to new releases when convenient for that project. It also allows application engineers to make local changes and bug fixes after framework development has been "frozen" for that project.

6.2.3 Keep the Model Stable

Backward compatibility is important with rapidly changing code. For frameworks this means the domain model must support the ways the application engineers use it. Application engineers hate releases with architectural changes. Although improving the model may require "code breaking" changes, try to keep them to a minimum. Find temporary ways to support "old style" objects to give engineers time to convert.

6.3 Developer Relations

In addition to developing a model and providing code for it, the framework team will spend a considerable amount of time supporting their customers, the application engineers.

6.3.1 Solicit Client Input

The framework team will increasingly lose touch with application development as it concentrates on the framework. Therefore, develop an ongoing dialog with framework clients (i.e., the project engineers). Goals for the framework will change over time. Some method of constantly reevaluating them should be available.

6.3.2 Promote Object-Oriented Skills

Even if the quality of the object model developed is excellent, application engineers will need object-oriented skills. In order to insure the success of the framework it may be appropriate to include object education as framework support.

6.3.3 Assist in Using the Framework Effectively

Even if application engineers have object-oriented experience, solutions to some problems may not be obvious. During framework development many domain problems will be considered and plans made to allow for their solution. If application engineers are not aware of these proposed solutions they may not use the framework effectively. Since the framework team will be the experts within the domain, take advantage of design reviews to assist application engineers with domain-related object design.

6.3.4 Refuse Ownership of Project-Specific Problems

When assisting project engineers, pressure will mount to use framework team members to solve project-specific problems. While this is most evident near the beginning of framework development, it will continue to plague the team years later. Resist the urge to allow framework engineers to participate on an application team. Framework members should be consulted on issues of the framework or for advice but application development should remain separate.

7. CONCLUSION

Developing a domain-specific, object-oriented framework has allowed ImageBuilder Software to remain competitive, grow and succeed in the fast-paced environment of software development. Providing applications within the framework domain has proven reliable and profitable over the past six years. While most of the techniques presented have a proven track record, some of the ideas only become clear in hindsight. Though ImageBuilder developed these recommendations through trial and error, we count the project as a success. Other organizations, with the benefit of these recommendations, should experience smoother sailing.

ACKNOWLEDGEMENTS

This paper was prepared for presentation at WICSA1. Thanks are due to Roger Bonzer for his many comments and Terry Hamm for providing the opportunity to write.

Event-Based Execution Architectures for Dynamic Software Systems

James Vera, Louis Perrochon, David C. Luckham
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, USA
{vera,perrochon,dcl}@pavg.stanford.edu

Key words: Evolutionary software architectures, software artifacts, component engineering.

Abstract: Distributed systems' runtime behavior can be difficult to understand. Concurrent, distributed activity make notions of global state difficult to grasp. We focus on the runtime structure of a system, its *execution architecture*, and propose representing its evolution as a partially ordered set of predefined *architectural* event types. This representation allows a system's topology to be visualized, analyzed and constrained. The use of a predefined event types allows the execution architectures of different systems to be readily compared.

1. INTRODUCTION

Distributed software systems consist of computational components interacting over a communications infrastructure. The executions of these systems can be highly dynamic with components being created and destroyed and the communications infrastructure undergoing continual reconfiguration. We propose to represent the evolution of the structure of such a running system, termed the *execution architecture* of the system, as a set of events, partially ordered by time and causality. This partial order of *architectural* events enables the precise analysis of the topological evolution of a system, just as a partial order of behavioral events enables a precise analysis of the functional activity of a system (Peled, Pratt et al. 1996).

The need for understanding execution architectures is driven by the main trends of software. *Component-oriented software engineering* has resulted in systems composed of components connected through middleware. *Distribution*, especially large scale, leads to asynchronous systems. The effect on execution architecture is dramatic: there may be no single depiction of the execution architecture of an asynchronous distributed system at a particular "point" in time. Instead, different observers can have a different views of what the architecture is.

We define a model for execution architectures and event types used to indicate changes in such a model. We show how systems such as distributed Java programs or systems communicating over commercial middleware can have their topological evolution projected onto our model. Using a predefined set of event types allows us to compare the execution architectures of systems implemented in different languages and which utilized different communications middleware.

Finally, we show how our representation of an execution architecture allows a system's topological evolution to be visualized, analyzed, and constrained.

2. PREVIOUS WORK

Our work is builds on two previously separate lines of research: software architecture and causal modeling.

2.1 Software Architecture

The term *architecture* has been widely discussed in the literature (e.g., (Garlan and Shaw 1993) (Moriconi and Qian 1994) (Perry and Wolf 1992) (Thompson 1998)). Soni et al. (Soni, Nord et al. 1995) discuss four categories of architecture: Conceptual, Module, Execution and Code. Conceptual architecture describes a system in terms of high level, abstract elements. Module architecture is the a more detailed functional decomposition. Execution architecture is the structure of the running system. Code architecture is the organizational structure of the source code of the system. Execution architecture is unique among the four in being a dynamic structure. We focus on execution architecture and argue that its appropriate representation is a partially ordered set of events.

Current research in software architectures has often focused on conceptual or module architectures (we will term architectures in either of these categories as *component* architectures). Architectures are described as entities possibly within other entities and interconnected somehow. Such descriptions are sometimes referred to as "boxes and arrows" representations. While being useful for many purposes, they have their shortcomings in describing a dynamic system. The representation of an execution architecture needs to be able to deal with change. In simple cases, execution architectures may be thought of as a series of static architectures, snapshots at different points in time. However, in many cases this is not enough.

The ACME system developed by Garlan et al. is designed as a language for exchanging architectural designs (Garlan, Monroe et al. 1995). The ACME system is inherently static though there is a proposed extension to allow the specification of potential dynamism. Darwin (Magee, Dulay et al. 1995) focuses on design specification and is not intended to be used in systems where new component types and the pathways between them are defined and added at runtime.

2.2 Causal Modeling

The use of partial orders of events to depict the behavior of distributed systems is well established (Lamport 1978; Pratt 1986). The relation of the partial order, typically called *causality*, enables true concurrency to be represented, information which is lost in a trace-based model.

Fidge and Mattern (Fidge 1988; Mattern 1988) separately developed the notion of vector time which is an algorithmic way of representing and analyzing the causal relation. Subsequent work has been done in improving the performance of such algorithms in special cases (e.g., (Meldal, Sankar et al. 1991). See (Schwarz and Mattern 1994) for an excellent survey). Other work has been done on applying causal modeling notions to existing programming languages (Santoro, Mann et al. 1998).

Our framework for execution architectures is an extension of our previous work in event-based systems (Luckham, Augustin et al. 1995; Luckham and Vera 1996). There we created a programming language, `RAPIDE`, in which a causal record of a program's behavior was automatically deduced and recorded during the program's execution.

3. A THEORY OF EXECUTION ARCHITECTURE

3.1 Execution Architectures

Execution architecture is a runtime notion. It is the architecture of an executing system. Its building blocks are executable constructs (e.g., objects, processes, tasks) which we call *modules* and the mechanisms they use to communicate which we call *pathways*. Both of these building blocks may be created and deleted during the system's execution making execution architecture an inherently dynamic notion. It can best be thought of as the record of the evolution of the structure of a running system.

3.2 Modules and Pathways

Our framework for execution architectures is built on two basic constructs:

1. *Modules* which are groupings of computational capabilities, and
2. *Pathways* which are the means modules use to communicate amongst themselves.

Module: A module is a grouping of computational capabilities. Modules have an associated type. The type consists of a set of provided and required features of each module, called *declarations*. These declarations are used to communicate with other modules. In an event-based system, these declarations would denote what events a module can send and receive. In a system based on synchronous (remote) procedure calls, the declarations would describe the procedures provided and called by each module. The type of a module describes what the module requires from other modules as well as what the module provides to other modules. Some architecture description language type systems only describe what modules provide.

In addition, we define a parent-child containment relationship over modules. Each module has maximum one parent. The parent relationship forms a directed graph. Being dynamic, the parent of a module may change. While parent-child is the only module relation we predefine, additional relationship may be defined, such as a relation between the software modules and the hardware modules they currently run on, etc.

Pathway: A pathway represents potential communication among modules. A pathway has a name, a set of inputs and a set of outputs. The inputs may be thought of as those things which invoke or use the pathway and the outputs as those things which result from the invocation or observe the use of the pathway. The inputs and outputs of a pathway may change. Typically, one input or output identifies a pair (module, declaration).

More generally, we allow the use of patterns to concisely specify sets of inputs or outputs. For example, a pattern could express "any module of type Airplane performing a RadioOut event." A pathway also has a scope over which it operates. The scope may be a particular module or the entire system. A pathway can represent a mechanism or simply a state or condition. Possible examples of pathways are a UNIX pipe, a Java socket, a serial cable between two computers, or a dynamic scoping rule of a particular programming language.

What constitutes a module is a subjective determination. For example, in a producer-consumer example, the producer and consumer are likely to be modules while the data communicated between them is probably not. Thus what is defined as objects in the source language does not necessarily correspond to modules. Not all objects need be modules, not all modules need be objects. In a system of workstations and network links one modeler may choose to have the workstations be represented as modules and the network links to be pathways. However, for a modeler more concerned with the network protocols, the network links might be the modules and the workstations the pathways. The key point is that modules represent the building blocks of the architecture. The definition of the actual correspondence is determined by the system implementor though language/system defaults may be used.

3.3 Execution Architecture Events

An execution architecture changes over time. Modules are created and destroyed, pathways come into and go out of existence. Such occurrences may be serialized or may happen independently. We model such changes as **events**. For example, the creation of a module or the additional of an output to a pathway would each be denoted by events. In our framework, we have templates for nine architectural events to describe creation and deletion of modules and pathways, addition and deletion of inputs and outputs from pathways, and changing of the parent of a module.

Events have parameters containing additional information. A `CreateModule`, for example, has parameters denoting the type of the module that was created, the parent of that module, and the name of the module. We give the simplified description of the templates below:

```
CreateModule(type : ModuleType, parent : Event,
             name : String);
DeleteModule(module : Event);
CreatePathway(inputs : Pattern, outputs : Pattern,
              name : String);
DeletePathway(pathway : Event);
ChangeParent(module : Event, parent : Event);
AddPathwayInputs(pathway : Event, inputs : Pattern);
AddPathwayOutputs(pathway : Event, outputs : Pattern);
DeletePathwayInputs(pathway : Event, inputs : Pattern);
DeletePathwayOutputs(pathway : Event, outputs : Pattern);
```

Some explanations may be necessary: first, our events do not directly refer to modules or pathways, as modules and pathways are transient objects in an execution architecture. In many cases, such as debugging post mortem, these objects no longer exist. Instead, we refer to the event that denotes the creation of the module or pathway. This can be seen in the parent parameter of `CreateModule` Instead of referring to the parent module, we refer to the `CreateModule`-event of the parent.

Second, we would like to be able to define the inputs and outputs of a pathway in a descriptive way, rather than as an enumeration of all possible inputs. `RAPIDE` allows us to easily describe the sets of input and output of a pathway using a *pattern*. For our purpose, the pattern in `CreatePathway` just specifies a set of declarations of certain modules. If a pattern language is not available, sets of pairs of a module (denoted by an event) and a declaration would work also.

Events are ordered temporally and causally. In the context of an event processing system such as `RAPIDE`, our architectural events can be treated like normal events. This allows us to use existing browsing tools and, more interestingly, pattern matching and constraint tools on architectural events. Using event constraint tools, we can write *topological* architecture constraints. Examples of such are presented in section 4.1.

3.4 Causal and Time Orders

When events are created they are (partially) ordered by cause and time. Two events are temporarily ordered if their temporal relation can be determined by any single clock in the system. The temporal order of two events in a distributed system without a common clock is not a priory known, but may be derived later. Two events are causally ordered if one causes the other (transitively). The exact meaning of *cause* is configurable and is captured by the system architect in a causal model. A common definition is that the events produced by a thread are totally ordered, the receipt of an event causally follows its sending.

The partial ordered set (poset) of architectural events forms a record of the evolution of the architecture. Recording relations between events in distributed systems as partial orders (instead of just time-stamping them) reveals that "the execution architecture at a certain point in time" is not a well defined concept. (Vera 1998) introduces the notion of **consistent cuts** as architectural observation points. A consistent cut partitions a partially ordered set into a *before* and *after* part. If an event is in the after part, then all events that follow it temporarily or causally are in the after part, and vice versa. Informally, an observer could have seen only and exactly the *before* part of the poset. When we speak of a "point" in the execution we mean "at a consistent cut".

3.5 Static Snapshots

At any consistent cut in the poset, a static representation or *snapshot* of the execution architecture similar to a component architecture may be derived from all of the events preceding the consistent cut. Such a snapshot is amenable to the types of analysis typically done on component architectures.

A *compatible sequence* of consistent cuts is graphically defined as a sequence of cuts which do not cross. Such a sequence may be viewed as an animated movie of the architecture's evolution. Since a poset may contain a set of such sequences, an execution architecture may contain a set of such animations. Each animation corresponds to a particular observers view of the architecture over time. The example below gives examples for such compatible and incompatible sequences of consistent cuts.

4. APPLICATIONS OF EXECUTION ARCHITECTURES

4.1 An Air Traffic Control System

Consider an air traffic control system as depicted in figure 1. Its architecture consists of AirTrafficSector which contains a ControlTower and a Runway-Control module.

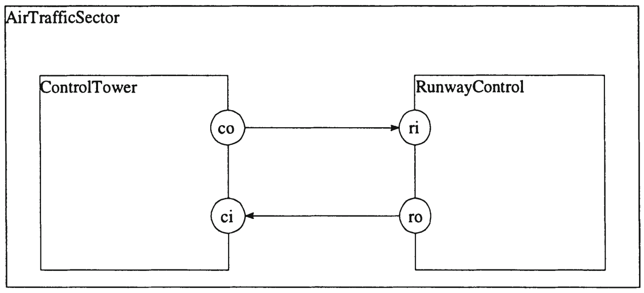


Figure 1. Initial air traffic architecture

This initial architecture was created by the execution represented by the poset in figure 2. The arrows denote the causal relation. Note that the consistent cut C1 in figure 2 marks the "point" in the execution at which the architecture depicted in figure 1 holds.

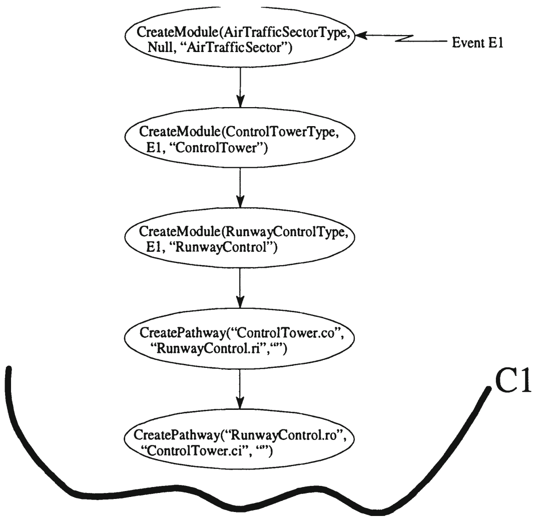

CreateModule(AirTrafficSectorType, Null, "AirTrafficSector")
Event E1
CreateModule(ControlTowerType, E1, "ControlTower")
CreateModule(RunwayControlType, E1, "RunwayControl")
CreatePathway("ControlTower.co", "RunwayControl.ri","")
CreatePathway("RunwayControl.ro", "ControlTower.ci", "")
C1

Figure 2. An initial execution of the air traffic system

Next imagine that two Flights (one called UA17, the other AA23) are created and that their creations are independent. A pathway from each Flight to the ControlTower is also created. This execution is represented by the poset in figure 3. At the point in that poset indicated by consistent cut C3 the architecture depicted in figure 4 holds.

In between consistent cut C1 and consistent cut C3 there are seven consistent cuts[1] two of which are shown in figure 3. Cuts C2a and C2b are inconsistent (graphically the cuts cross) so they would not both appear in the same architecture animation. One architecture animation A1 could consist of sequence of consistent cuts C1, C2a, C3 and another architecture animation A2 could consist of the sequence C1, C2b, C3.

In architecture animation A1, the initial snapshot shown in figure 1 would appear, then Flight UA17 and its connection to the ControlTower would appear and finally Flight AA23 and its connection to the ControlTower would appear. In architecture animation A2, the same initial architecture as in A1 would appear, followed by the appearance of Flight

[1]The consistent cut for which the maxima is (1) Event E6, (2) Event E7, (3) Event E8 (this cut is labeled C2a in figure 3, (4) Event E9 (this cut is labeled C2b in figure 3, (5) Events E6 and E7, (6) Events E6 and E9 and (7) Events E7 and E8

AA23 and its connection to the ControlTower followed by Flight UA17 and its connection to the ControlTower.

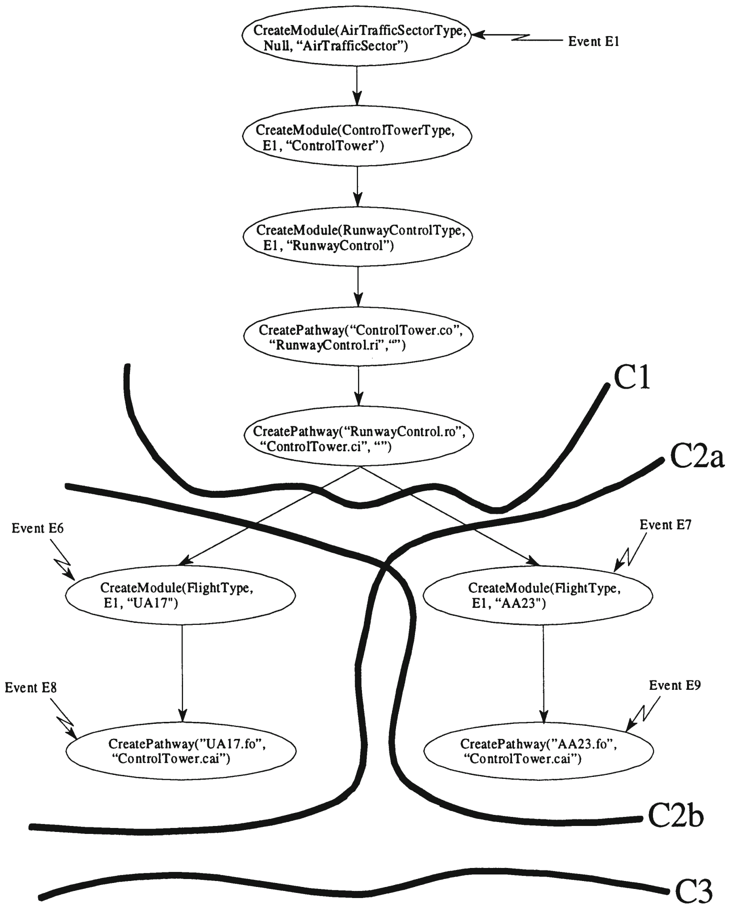

CreateModule(AirTrafficSectorType, Null, "AirTrafficSector")
Event E1
CreateModule(ControlTowerType, E1, "ControlTower")
CreateModule(RunwayControlType, E1, "RunwayControl")
CreatePathway("ControlTower.co", "RunwayControl.ri","")
CreatePathway("RunwayControl.ro", "ControlTower.ci", "")
C1
C2a
Event E6
Event E7
CreateModule(FlightType, E1, "UA17")
CreateModule(FlightType, E1, "AA23")
Event E8
Event E9
CreatePathway("UA17.fo", "ControlTower.cai")
CreatePathway("AA23.fo", "ControlTower.cai")
C2b
C3

Figure 3. Continuation of execution of the air traffic system

In a system which is merely time-stamping its architectural changes, or which observes them by breakpointing the system, only animation A1 or animation A2 would be seen (or potentially a third animation A3 in which at one "frame" neither flight is visible and in the next both are. This animation would result from an overly coarse time-stamping or breakpointing interval.) This is a specific instance of a more general case. Whenever there are concurrent changes to an architecture, a single trace of those changes (such as would result from time-stamping or breakpointing) will only capture one animation. They cannot capture the information contained in incompatible consistent cuts.

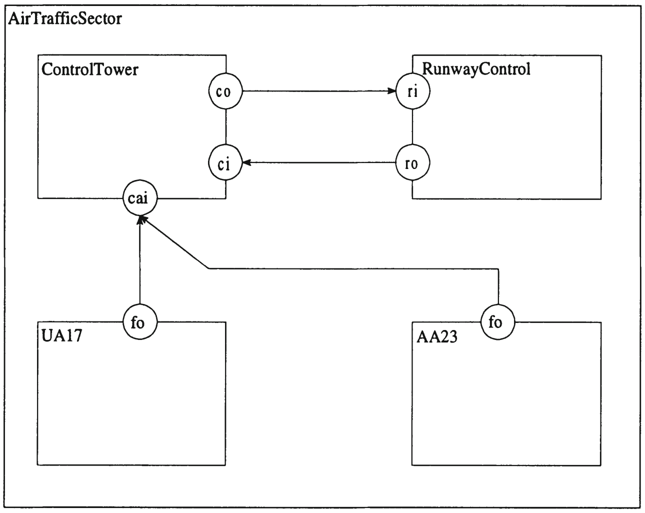


Figure 4. Air traffic architecture at consistent cut C3

4.1.1 Use of Partially Ordered Architectural Events

The representation of execution architecture as partially ordered sets (posets) of events allows poset oriented tools and methods to be applied to execution architectures. In particular, the pattern and constraint languages developed in the RAPIDE project may be applied to specify topological (as opposed to purely functional) constraints on executing systems. The RAPIDE languages can be used to set up simple filters, constraints or maps. Some illustrative examples follow.

4.1.2 Filters

Filters are operators which take as input a poset and output a subset of the input selected by a pattern. Filters allow a reduction of the space being examined. Suppose we are only interested in the module containment structure. The following filter could be used:

```
observe select CreateModule() or DeleteModule()
            or ChangeParent();
```

4.1.3 Constraints

The representation of an execution architecture as a poset allows us to write constraints about its evolution as well as static snapshots. For example, we could constrain that a Radar module must be created before a Depot module. Or that a particular communication topology (full connected, strongly connected) exists among a class of modules before some condition.

Given a poset constraint language such as that available in RAPIDE, the existence of architecture events allows these specifications of topological constraints. In an event generating system (where the behavior is also represented as events), mixed-mode functional/topological constraints can be expressed.

Suppose we want to require that the creation of Flight modules be serialized. We might make this requirement because the creation of a new Flight module involves the manipulation of some global state (e.g., the number of Flights currently in the sector). We can express this constraint as requiring the events signifying the creation of Flight modules be totally ordered:

```
observe select CreateModule(type is FlightType)
 match [* rel -> ] CreateModule;
```

4.1.4 Maps

Maps are operators that transform a poset into a new poset. The new poset is generally at a higher level of abstraction. That allows the behavior of a system to be understood in more abstract terms than those in which it was implemented.

As a simple example, suppose we wish to abstract ControlTower module and RunwayControl module pairs into a single AirportControl module. To do this we would create a map that does this abstraction and adjusts the communication structure accordingly. If the input poset also contained the functional behavior of the system then behavior of a ControlTower or RunwayControl module would also need to be mapped into behavior by an AirportControl module. A subset of such a map is given below:

```
map AirportAbstract is trans : array [Event] of Event;
    (?c,?r,?p, ?a : Event; ?s1,?s2 : String)
    ?c@CreateModule(ControlTowerType, ?p, ?s1) and
    ?r@CreateModule(RunwayControlType, ?p, ?s2)
=>  ?a@CreateModule(AirportControlType, ?p, ?s1+?s2);
    trans[?c] := ?a; trans[?r] := ?a; end map;
```

The above rule looks for pairs of CreateModule events, one denoting the creation of a ControlTower module, the other a RunwayControl module. If they both have the same parent then a CreateModule event is created in the new poset which denotes the creation of an AirportControl module. The association of the lower level events to the higher level event is stored in an associative array for subsequent use by other rules.

4.1.5 Conformance to Reference Architectures

By combining maps and constraints, the conformance of systems to reference architectures may be checked (Luckham, Augustin et al. 1995). Architecture events allow topological conformance to be expressed. This can be useful for checking requirements such as duplicate communication channels.

4.1.6 Reverse Engineering

Reverse engineering of architectures is necessary when the original architecture has been lost (or never existed). Research has focused on extracting component architectures from source code (Harris, Reubenstein et al. 1995). By extracting architecture events from a running system via instrumentation (such as monitoring middleware) we can extract the execution architecture even when the original source code is unavailable. Perhaps more comparative work is the extraction of call trees by debugging software. These tools can be thought of as providing a maximal depiction of the use of the execution architecture. An execution architecture poset, in contrast, captures its evolution.

4.2 Applications to Other Domains

The mapping of concepts from event-based systems into our architectural constructs is flexible and in each case, different strategies are supported with emphasis on different attributes.

Whatever choice is made, the ability to map one poset into another allows such decisions to be changed ex post facto. In the above example, the choice of the assignment to modules and pathways could be inverted by a mapping.

In this subsection we present some example translations of distributed systems to our execution architecture constructs.

4.2.1 A System Implemented in Java

The Java notion of objects is easily mapped to our module concept. More interesting is the choice of constructs which map to pathways. The ability of one object to name another object (generally known as dynamic scoping) is one form of pathway. If an object A can name an object B then we can say that a pathway exists from A to B.

The Java socket construct is amenable to translation into a pathway. A Java socket is a bidirectional mechanism over which data may be sent from one object to another. It has two ends. Any object which can name an end may send or receive data along the socket. Therefore, a Java socket could be translated into two of our pathway constructs (pathways are one directional while sockets are bidirectional) where the sources of one of the pathways are the destinations of the other (and vice versa).

4.2.2 A System Hosted on Commercial Middleware

The Information Bus (TIB) (TIBCO 1998) is a communication middleware which supports the subject-based publish-subscribe metaphor. Objects send out (publish) messages labeled with a particular textual field (subject). Other objects can request to receive (subscribe to) messages with a particular subject. Higher level protocols are built on top of the publish/subscribe mechanism such as point to point communication, synchronous communication, and automatic selection of one from several destinations.

In our application of execution architectures to the TIB (Luckham and Frasca 1998), we map each TIB client into a module and map the basic publish/subscribe mechanism into pathways. In a component architecture description, for every subject a connection is needed from the modules which may publish that subject to the modules that may subscribe to the subject. Not surprisingly, pictures of such architectures show the TIB only as a bus. In an execution architecture, pathways are only maintained between modules that actually publish and modules that actually listen to a certain subject, e.g., only after a module subscribes to a subject it is added as a destination of the pathway which corresponds to that subject. This results in a point-to-point depiction of the communication network.

Other TIB protocols can be captured via their implementation on top of the publish/subscribe protocol. However, the semantics of the higher level protocols are more accurately captured by dealing with them individually.

5. SUMMARY AND CONCLUSIONS

We developed a technology to define, track and control execution architectures of dynamically changing software systems. Architectural changes are represented by causally and temporarily (partially) ordered events. Our framework has the following features:

- Architecture events provide a formal language to describe execution architectures.
- Filters and maps, together with the visualization tools such as those available in RAPIDE allow real time monitoring of execution architectures.
- The RAPIDE engine raises exception when the formal specification (i.e., constraints) of an execution architecture is violated. Maps allow corrective actions in non fatal error conditions.
- Static snapshots at consistent cuts provide backward compatibility with previous approaches.
- Posets of architecture events capture the execution architecture of an asynchronous, distributed system in cases where static architectures are not expressive enough.
- Posets can easily be stored and analyzed at a later time.

Our technology is applicable to systems that are distributed, asynchronous and have a high change rate. We believe understanding execution architectures is important because it fills the gap between the abstractness of conceptual architectures and what is actually implemented in systems. In particular, our partially-ordered event-based execution architectures is superior to simple, time-stamped traces of architectural changes.

REFERENCES

Fidge, C. J. (1988). Partial Orders for Parallel Debugging. Workshop on Parallel and Distributed Debugging, Madiscon, Wisconsin, ACM SIGPLAN/SIGOPS.

Garlan, D., R. Monroe, et al. (1995). ACME - Software Architecture Interchange Language.

Garlan, D. and M. Shaw (1993). An Introduction to Software Architecture, World Scientific Publishing Company.

Harris, D. R., H. B. Reubenstein, et al. (1995). Reverse Engineering to the Architectural Level. 17th International Conference on Software Engineering, ACM.

Lamport, L. (1978). "Time, Clocks, and the Ordering of Events in a Distributed System." CACM 21(7): 558-565.

Luckham and Vera (1996). "An Event-Based Architecture Definition Language." IEEE Transactions on Software Engineering 21(9): 717-734.

Luckham, D. C., L. M. Augustin, et al. (1995). "Specification and Analysis of System Architectures using RAPIDE." IEEE Transactions on Software Engineering 21(4).

Luckham, D. C. and B. Frasca (1998). Complex Event Processing in Distributed Systems. Stanford, Stanford University.

Magee, J., N. Dulay, et al. (1995). Specifying Distributed Software Architectures. 5th European Software Engineering Conference (ESEC 95), Sitges, Spain.

Mattern, F. (1988). Virtual Time and Global States of Distributed Systems. Parallel and Distributed Algorithms, Elsevier Science Publishers.

Meldal, S., S. Sankar, et al. (1991). Exploiting Locality in Maintaining Potential Causality. 10th ACM Symposium on the Principles of Distributed Computing, New York, New York, ACM Press.

Moriconi, M. and X. Qian (1994). Correctness and Composition of Software Architectures. SIGSOFT'94 Software Engineering Notes, New Orleans, LA, ACM Symposium on Foundations of Software Engineering.

Peled, D. A., V. R. Pratt, et al. (1996). Partial Order Methods in Verification, American Mathematical Society.

Perry, D. E. and A. L. Wolf (1992). Foundations for the Study of Software Architecture, SIGSOFT '92, Software Engineering Notes, ACM Symposium on Foundations of Software Engineering.

Pratt, V. R. (1986). "Modeling concurrency with partial orders." Int. J. of Parallel Programming 15(1): 33-71.

Santoro, A., W. Mann, et al. (1998). eJava - Extending Java with Causality. 10th International Conference on Software Engineering and Knowledge Engineering (SEKE'98), Redwood City, CA, USA.

Schwarz, R. and F. Mattern (1994). "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail." Distributed Computing 7(3): 149-174.

Soni, D., R. L. Nord, et al. (1995). Software Architecture in Industrial Applications. 17th International Conference on Software Engineering, ACM.

Thompson, C., Ed. (1998). Workshop on Compositional Software Architectures. Monterey, California, OMG, DARPA, MCC, OBJS.

TIBCO (1998). TIBCO Web Site, TIBCO.

Vera, J. S. (1998). Software Architecture Description Languages: Descriptive Constructs and Execution Algorithms. Electrical Engineering. Stanford, Stanford University.

DOMAIN-SPECIFIC ARCHITECTURES AND PRODUCT FAMILIES

Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study

Jan Bosch
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: Jan.Bosch@ide.hk-r.se
www:http://www.ide.hk-r.se/~bosch

Key words: Reusable assets, product-line architectures, software composition, software evolution, case study

Abstract: In this paper, a case study investigating the experiences from evolution and modification of reusable assets in product-line architectures is presented involving two Swedish companies, Axis Communications AB and Securitas Larm AB. Key persons in these organisations have been interviewed and information has been collected from documents and other sources. The study identified problems related to multiple versions of reusable assets, dependencies between assets and the use of assets in new contexts. The problem causes have been identified and analysed, including the early intertwining of functionality, the organizational model, the time to market pressure, the lack of economic models and the lack of encapsulation boundaries and required interfaces.

1. INTRODUCTION

Product-line architectures have received attention in research, but even more so in industry. Many companies have moved away from developing software from scratch for each product and instead focused on the commonalities between the different products, and capturing those in a product-line architecture and an associated set of reusable assets. This is, especially in the Swedish industry, a logical development since software is an increasingly large part of products and often defines the competitive advantage. When moving from a marginal to a major part of products, the required effort for software development also becomes a major issue and industry searches for ways to increase reuse of existing software to minimize product-specific development and to increase the quality of software.

A number of authors have reported on industrial experiences with product-line architectures. In [SEI 97], results from a workshop on product line architectures are presented. Also, [Macala et al. 96] and [Dikel et al. 97] describe experiences from using product-line architectures in an industrial context. The aforementioned work reports primarily from large, American software companies, often defense-related, which are not necessarily representative of the software industry as a whole, especially European small- and medium-sized enterprises.

We have performed a case study of product-line architectures involving two Swedish software development organisations: Axis Communications AB and Securitas Larm AB. The former develops and sells network-based products, such as printer, scanner, camera, and storage servers, whereas the latter company produces security- and safety-related products such as fire-alarm, intruder-alarm, and passage control systems. Since the beginning of the '90s, both organisations have moved towards product-line architecture based software development, especially through the use of object-oriented frameworks as reusable assets. In an earlier paper [Bosch 98c], we reported on the technological, process, organizational and business problems and issues related to product-line architectures. In this paper, we focus on the use, evolution, composition and reuse of assets that are part of a product-line architecture. Since the involved organisations have considerable experience using this approach, we report on their way of organising software development, the obtained experiences and the identified problems.

The contribution of this paper is, we believe, its provision of exemplars of industrial organisations in software industry that can be used for comparison or as inspiration. In addition, the experiences and problems surrounding reusable assets provide, at least partly, a research agenda for the software architecture and software reuse communities.

The remainder of the paper is organised as follows. In the next section, the research method used for the case study is briefly described. The two companies forming the focus of the case study are described in section 3. Section 4 discusses the differences in perception of product-line architectures and reusable assets in academia and industry. The problems identified during data collection are discussed in section 5 and their causes are analysed in section 6. Section 7 discusses related work and the paper concludes in section 8.

2. CASE STUDY METHOD

The goal of the study was twofold: first, our intention was to get an understanding of the problems and issues surrounding reusable assets part that are part of a product-line architecture in "normal" software development organisations, i.e., organisations of small to average size, i.e., tens or a few hundred employees, and unrelated to the defense industry. Second, our goal was to identify those research issues that are most relevant to software industry with respect to reusable assets in product-line software architectures.

The most appropriate method to achieve these goals, we concluded, was interviews with the system architects and technical managers at software development organisations. Since this study marks the start of a three year government-sponsored research project on software architectures involving our university and three industrial organisations, i.e., Axis Communications, Securitas Larm, and Ericsson Mobile Communications, the interviewed parties were taken from this project. The third organisation, a business unit within Ericsson Mobile Communications, is a recent start-up and has not yet produced product-line architectures or products. A second reason for selecting these companies was that we believe them to be representative of a larger category of software development organisations. These organisations develop software that is to be embedded in products also involving hardware and mechanics, are of average size (e.g., development departments of 10 to 60 engineers), and develop products sold to industry or consumers.

The interviews were open and rather unstructured, although a questionnaire was used to guide the process. The interviews were video-taped for further analysis afterwards and in some cases documentation from the company was used to complement the interviews. The interviews often started with a group discussion and were later complemented with interviews with individuals for deeper discussions on particular topics.

3. CASE STUDY ORGANISATIONS

3.1 Case 1: Axis Communications AB

Axis Communications started its business in 1984 with the development of a printer server product that allowed IBM mainframes to print on non-IBM printers. Up to then, IBM maintained a monopoly on printers for their computers, with consequent price settings. The first product was a major success that established the base of the company. In 1987, the company developed the first version of its proprietary RISC CPU that provided better performance and cost-efficiency than standard processors for their data-communication oriented products. Today, the company develops and introduces new products on a regular basis. At the beginning of the '90s, object-oriented frameworks were introduced into the company and since then a base of reusable assets is maintained from which most products are developed.

Axis develops IBM-specific and general printer servers, CD-ROM and storage servers, network cameras, and scanner servers. The latter three products, in particular, are built using a common product-line architecture and reusable assets. In figure 1, an overview of the product-line and product architectures is shown. The organisation is more complicated than the standard case with one product-line architecture (PLA) and several products below this product-line. In the Axis case, there is a hierarchical organisation of PLAs, with the product-line architecture at the top and the product-group architectures (e.g., the storage-server architecture) at the next lower level. The focus of the case study is on the marked area in the figure, although the other parts are discussed briefly as well. The primary reusable assets for Axis include object-oriented frameworks for file systems and network protocols, but several smaller frameworks are used as well.

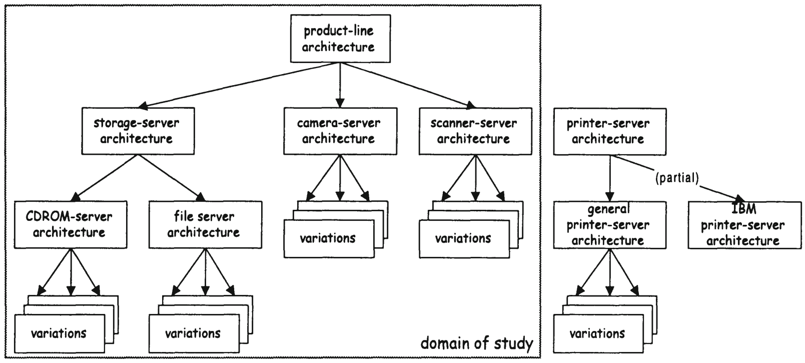

product-line architecture
storage-server architecture
camera-server architecture
scanner-server architecture
printer-server architecture
(partial)
CDROM-server architecture
file server architecture
variations
variations
general printer-server architecture
IBM printer-server architecture
variations
variations
domain of study
variations

Figure 1. Product-line and product software architectures in Axis Communications

3.2 Case 2: Securitas Larm AB

Securitas Larm AB (formerly TeleLarm AB) develops, sells, installs and maintains safety and security systems such as fire-alarm systems, intruder alarm systems, passage-control systems, and video surveillance systems. The company's focus is especially on larger buildings and complexes, requiring integration between the aforementioned systems. Therefore, Securitas has a fifth product unit developing integrated solutions for customers including all or a subset of these systems. In figure 2, an overview of the products is presented.

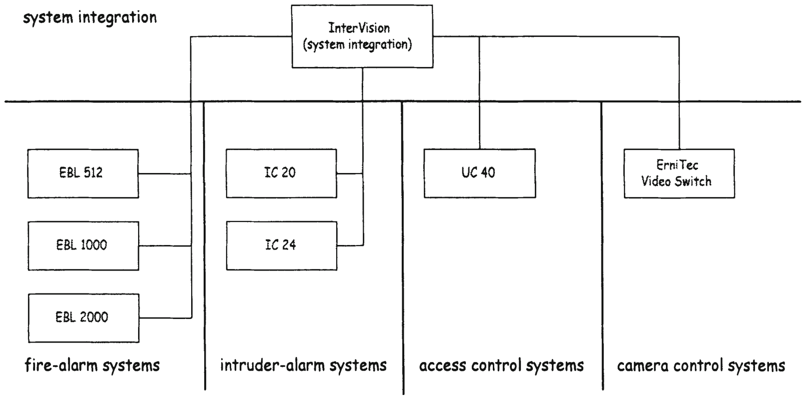


Figure 2. Securitas Larm Product Overview

Securitas uses a product-line architecture only for their fire-alarm products, in practice only the EBL 512 product, and traditional approaches in the other products. However, due to the success in the fire-alarm domain, the intention is to expand the PLA in the near future to include the intruder-alarm and passage-control products as well.

4. PRODUCT-LINE ARCHITECTURES AND REUSABLE ASSETS

An important issue we identified during this case study and our other cooperative projects with industry is that there exists a considerable difference between the academic perception of software architecture and reusable assets and the industrial practice. It is important to explicitly discuss these differences because the problems described in the next section are based on the industrial rather than the academic perspective. It is interesting to note that sometimes the problems that are identified as the most important and difficult by industry are not identified (or viewed as non-problems) by academia.

Table 1 lists the academic and industrial interpretations of the notion of product-line architecture. The main differences are related to the definition of architectures, the use of first-class connectors, and the use of specialised languages.

Table 1. Academic versus industrial view of software architecture

Research	Industry
Architecture is explicitly defined.	Mostly conceptual understanding of architecture. Minimal explicit definition, often through notations.
Architecture consists of components and first-class connectors.	No explicit first-class connectors (sometimes ad-hoc solutions for run-time binding and glue code for adaptation between assets).
Architectural description languages (ADLs) explicitly describe architectures and are used to automatically generate applications.	Programming languages (e.g., C++) and script languages (e.g., Make) used to describe the configuration of the complete system.

For reusable assets, one can identify a similar difference between the academic and industrial understanding of the concepts. In table 2, an overview is presented comparing the two views. The main differences are related to, among others, the assumed black-box nature, the component interface, and variability.

Table 2. Academic versus industrial view of reusable assets

Research	Industry
Reusable assets are black-box components.	Assets are large pieces of software (sometimes more than 80 KLOC) with a complex internal structure and no enforced encapsulation boundary, e.g., object-oriented frameworks.
Assets have narrow interface through a single point of access.	The asset interface is provided through entities, e.g., classes in the asset. These interface entities have no explicit differences to non-interface entities.
Assets have few and explicitly defined variation points that are configured during instantiation.	Variation is implemented through configuration and specialisation or replacement of entities in the asset. Sometimes multiple implementations (versions) of assets exist to cover variation requirements
Assets implement standardized interfaces and can be traded on component markets.	Assets are primarily developed internally. Externally developed assets go through considerable (source code) adaptation to match the product-line architecture requirements.
Focus is on asset functionality and on the formal verification of functionality.	Functionality and quality attributes, e.g., performance, reliability, code size, reusability and maintainability, have equal importance.

5. PROBLEMS

Based on the interviews and other documentation collected at the organisations part of this case study, we have identified a number of problems related to reusable assets that we believe to have relevance in a wider context than just these organisations. In the remainder of this section, the problems that were identified during the data collection phase of the case study are presented. For each problem, a problem description is presented, illustrated by an example from one of the case-study companies. The problems are categorized into three categories, related to multiple versions of assets, dependencies between assets, and the use of assets in new contexts.

5.1 Multiple versions of assets

Product-line architectures have associated reusable assets that implement the functionality of architectural components. These assets can be very large and contain up to a hundred KLOC or more. Consequently, they represent considerable investments (multiple man-years in certain cases). Therefore, it was surprising to identify that in some cases, the interviewed companies maintained multiple versions (implementations) of assets in parallel. One can identify at least four situations where multiple versions are introduced.

5.1.1 Conflicting quality requirements

The reusable assets are generally optimized for particular quality attributes such as performance or code size. Different products in the product line, even though they require the same functionality, may have conflicting quality requirements. These requirements may have so high a priority that no single component can fulfil them all. The reusability of the affected asset is then restricted to just one or a few of the products while other products require another implementation of the same functionality.

For example, in Axis, the printer server product was left out of the product-line architecture (although it can be considered to be a PLA on its own, with more than 20 major variations) because minimizing the binary code size is the driving quality attribute for the printer server whereas performance and time to market are the driving quality attributes for the other network-server products.

Our impression is that when products in the product-line are at different points in their lifecycle, there is a tendency to have multiple versions of assets. This is because the driving quality attributes of a product tend to change during its lifecycle from feature- and time-to-market driven to cost- and efficiency-driven (see also [SEI 97]).

5.1.2 Variability implemented through versions

Certain types of variability are difficult to implement through configuration or compiler switches since the effect of a variation spreads out throughout the reusable asset. An example is different contexts, such as the operating system, for an asset. Although it might be possible to implement all variability through, for example, #ifdef statements, often it is decided to maintain two different versions.

The above printer server example can also be used here. The different versions of assets actually implement different variability selections.

5.1.3 High-end versus low-end products

The reusable asset should contain all functionality required by the products in the product-line, including the high-end products. The problem is that low-end products, generally requiring a restricted subset of the functionality, pay for the unused functionality in terms of code size and complex interfaces. Especially for embedded systems where the hardware costs play an important role in the product price, the software engineers may be forced to create a low-end, scaled-down version of the asset to minimize the overhead for low-end products.

Two versions of the file-system framework have been used in Axis in different products. The scanner and camera products used a scaled down version of the file system framework, only implementing a memory-based pseudo file system, whereas the CD-Rom and Jaz drive products used the full-scale file system, implementing a variety of file-system standards. The scanner and camera product develpoers had no interest in incorporating the complete asset since it required more memory than strictly necessary, leading to increased product cost.

5.1.4 Business unit needs

Especially in the organizational model used by Axis, where the business units are responsible for asset evolution, assets are sometimes extended with very product-specific code, or code only tested for one of the products in the product-line. The problems caused by this create a tendency within the affected business units to create their own copy of the asset and maintain it solely for their own product. This minimizes the dependency on the shared product-line architecture and solves the problems in the short term, but in the long term it generally does not pay off. We have seen several instances of cases where business units had to rework considerable parts of their code to incorporate a new version of the evolved shared asset that contained functionality that needed to be incorporated in their product also.

The aforementioned file system framework example is also an example of a situation where business-unit needs resulted in two versions of an asset. At a later stage, the full-scale file system framework had evolved and the scanner and camera products wanted to incorporate the additional functionality. In order to achieve that, the product-specific code of both products had to be reworked in order to incorporate the evolved file system framework.

5.2 Dependencies between assets

Since the reusable assets are all part of a product-line architecture, they tend to have dependencies between them. Although dependencies between assets are necessary, assets often have dependencies that could have been avoided by another modularization of the system or a more careful asset design. From the examples at the studied companies, we learned that the initial design of assets generally defines a small set of required and explicitly defined dependencies. It is often during evolution of assets that unwanted dependencies are created. Addition of new functionality may require extension of more than one asset; in the process dependencies are often created between the assets that implement the functionality. These new dependencies could often have been avoided by another decomposition of the architecture. They have a tendency to be implicit, in that their documentation is often minimal, and the software engineer encounters the dependency late in the development process. Dependencies in general, but especially implicit dependencies, reduce the reusability of assets in different contexts, but also complicate the evolution of assets within the PLA since each extension of one asset may affect multiple other assets. Based on our research at Axis and Securitas, we have identified three situations where new, often implicit, dependencies are introduced:

5.2.1 Component decomposition

With the development of the product-line architecture, generally the sizes of the reusable assets also increase. Companies often have some optimal size for an asset component, so that it can be maintained by a small team of engineers (e.g., it captures a logical piece of domain functionality, etc.). With the increasing size of asset components, there is a point where a component needs to be split into two components. These two components, initially, have numerous relations to each other, but even after some redesign several dependencies often remain because the initial design did not modularize the behaviour of by the two components. One could, obviously, redesign the functionality of the components completely to minimize the dependencies, but the required effort is generally not feasible in development organizations.

To give an example from Axis: at some point, it was decided that the file system asset should be extended with functionality for authorisation. To implement this, it proved to be necessary to also extend the protocol asset with some functionality. This created yet another dependency between the file system and the protocol assets, making it harder to reuse them separately. Currently, the access functionality has been broken out of the file system and protocol assets, and defined as a separate asset, but some dependencies between the three assets remain.

5.2.2 Extensions cover multiple assets

Extension of the product-line architecture stems from new functional requirements that need to be incorporated in the existing functionality. Often, the required extension to the product line covers more than one asset. During implementation of the extension, it is very natural to add dependencies between the affected assets since one is working on functionality that is perceived as one piece, even though it is divided over multiple assets.

The authorisation access extension to the Axis PLA provides, again, an excellent example. At first, the access functionality was added to the file system and protocol assets. However, the protocol framework contained the protocol user classes that were needed by the access functionality in the file system framework, leading to strong dependencies between the two frameworks. At a later stage, the authorisation access was separated from the two assets and represented as a single asset, thereby decreasing the dependencies.

5.2.3 Asset extension adds dependency

As mentioned, the initial design of a PLA generally minimizes dependencies between its components. Evolution of an asset component may cause this component to require information from an earlier unrelated component. If this dependency had been known during the initial PLA design, then the functionality would have been modularized differently and the dependency would have been avoided.

In the protocol framework in the Axis PLA, most of the implemented protocols use a layered organisation in which process packets that are sent up and down the protocol layers. These communication packets are nested in the sense that each lower-level protocol layer declares a new packet and adds the received packet as an argument. At some point, the implementation of new functionality required methods of the most encapsulated packet object to refer to data in one of the packets higher up in the encapsulation hierarchy, introducing a very unfortunate dependency between the two packets.

5.3 Assets in new contexts

Since assets represent considerable investments, the goal is to use assets in as many products and domains as possible. However, a new context differs in one or more aspects from the old context, causing a need for the asset to be changed in order to fit. Two main issues in the use of assets in new contexts can be identified.

5.3.1 Mixed behaviour

An asset is developed for a particular domain, product category, operating context, and set of driving quality requirements. Consequently, it often proves to be hard to apply the asset in different domains, products, or operating contexts. The design of assets often hard-wires design decisions concerning these aspects unless the type of variability is known and required at design time.

The main asset for Securitas is the highly successful fire-alarm system. In the near future, Securitas intends to develop a similar asset for the domain of intruder-alarm systems. Since the domains have many aspects in common, their intention is to reuse the fire-alarm asset and apply it to the intruder alarm domain, rather than developing the asset from scratch. However, initial investigations show that the domain change for the asset is not a trivial endeavour.

5.3.2 Design for required variability

It is recommended best practice that reusable assets be designed to support only the variability requested in the initial requirement specification, e.g., [Jacobson et al. 97]. However, a new context for a reusable asset often also requires new variability dimensions. One cannot expect that assets are designed to include all foreseeable forms of variability, but they should be designed so that the introduction of new variability requires minimal effort.

The application of the fire-alarm framework in the intruder-alarm domain serves as an example here. These systems share, to a large extent, the same operating context and quality requirements. However, since the fire-alarm domain functionality is hard-wired in the framework design, and the intruder alarm domain has different requirements and concepts, one is forced to introduce variability for application-domain functionality.

6. CAUSE ANALYSIS

The problems discussed in the previous section represent an overview of the issues surrounding the use of reusable assets in a product-line architecture. We have analysed these problems in their industrial context and have identified what we believe to be the primary underlying causes of these problems. In the remainder of this section, these causes are discussed.

6.1 Early intertwining of functionality

The functionality of a reusable asset can be categorized into functionality related to the application domain, the quality attributes, the operating context, and the product-category. Although these different types of functionality are treated separately at design time, both in the design model and the implementation they tend to be mixed. Hence it is generally hard to change one of the functionality categories without extensive reworking of the asset. Both the state-of-practice as well as leading authors on reusable software (e.g., [Jacobson et al. 97]), design for required variability only. That is, only the variability known at asset-design time is incorporated in the asset. Since the requirements evolve constantly, requirement changes related to the domain, product category, or context generally appear after design time. Consequently, it often proves hard to apply the asset in the new environment.

The early intertwining of functionality is a primary cause of several of the problems discussed in the previous section. Multiple versions of assets are required because the different categories of functionality cannot be separated in the implementation and implemented through variability. Also, the use of an asset in a new context is complicated by the mixing of functionality.

Companies try to avoid mixed functionality primarily through design. For instance, the use of layers, even in asset design, to separate operating-context-dependent from context-independent functionality, avoids the mixing. Also, several design patterns [Gamma et al. 94, Buschmann et al. 96] support separation of different types of functionality and support the introduction of variability.

Research issues. The primary research issue is to find approaches that allow for late composition of different types of functionality. Examples of this can be found in the Draco system [Neighbors 89], [Batory & O'Malley 92] approach to hierarchical software systems, parameterized programming [Goguen 96], aspect-oriented programming [Kiczales et al. 97] and in the layered object model [Bosch 98a] and [Bosch 98b]. In addition, design solutions, such as design patterns, that successfully separate functionality should be a continuing topic of research.

6.2 Organization

Both Securitas and Axis have explicitly decided against the use of separate domain engineering units. The advantages of separate domain engineering units, such as being able to spend considerable time and effort on thorough designs of assets, were generally recognised. On the other hand, people felt that a domain engineering group could easily get lost in wonderfully high abstractions and highly reusable code that did not quite fulfil the requirements of the application engineers. In addition, having explicit groups for domain and application engineering requires a relatively large software development department consisting of at least fifty to a hundred engineers.

Nevertheless, several of the problems discussed earlier can be related to the lack of independent domain engineering. Business units focus on their own quality attributes and design for achieving those during asset extension. Because of that, multiple versions of assets may be created where a domain engineering unit might have found solutions allowing for a single version. In addition, asset extension without sufficient focus on the product-line as a whole may introduce more dependencies than strictly necessary, complicating the use of assets as well as their reuse in new contexts.

Solutions exist to minimize the negative effects of organizational structures. At Axis, so-called asset redesigns are performed when a consensus is present that an asset needs to be reorganised. During an asset redesign, the software architects from the business units using the asset gather to redesign the asset in order to improve its structure. As a complement, both Axis and Securitas have responsibility for each asset, and evolution of assets has to be approved by them. However, because of time-to-market pressures, there is sometimes a need to accept less-than-optimal solutions. Thirdly, to improve on these issues, management must be willing to occasionally relieve some time-to-market pressure, accepting delay of one product so that subsequent products can enter the market sooner.

Research issues. The primary research issue concerns the processes surrounding asset evolution. More case studies and experimentation are required to gather evidence of working and failing processes, and mandatory and optional steps. In addition, one can conclude that it is unclear when an organisation should have separate domain engineering units rather than performing asset development in the application engineering units. Research is required for the collection of evidence on optimal organizational structures and identification and evaluation of approaches to minimize the negative effects of organizational choices.

6.3 Time to market

A third important cause for the problems related to reusable assets at the interviewed companies is the time-to-market (TTM) pressure. Getting out new products and subsequent versions of existing products is very high up on the agenda, thereby sacrificing other topics. The problem most companies are dealing with is that products appearing late on the market will lead to diminished market share or, in the worst case, to no market penetration at all. However, this all-or-nothing mentality leads to an extreme focus on short-term goals, while ignoring long term goals. Sacrificing some time-to-market for one product may lead to considerable improvements for subsequent products, but this is generally not appreciated.

The TTM pressure causes several of the problems discussed earlier. This is primarily because software engineers do not have the time to reorganise the assets to minimize dependencies or to generalize functionality. Asset evolution is often implemented as quick fixes, thereby decreasing the usability of the asset in future contexts.

To address the problems resulting from TTM pressure, it is important for software development organizations to regard the development of a product-line architecture and associated assets as a strategic issue, with decisions being made at the appropriate level. The consequences for the time-to-market of products under development should be balanced against the future returns. Finally, taking a time-out for asset redesign is necessary periodically to "clean up."

Research issues. Decisions related to TTM for products are made based on a business case and these, rather relevant, research issues are outside the software engineering domain. However, two issues can be identified: the lack of economic models (described in the next section) and design techniques that minimize the effort required for extending assets without diminishing their future applicability.

6.4 Economic models

As mentioned earlier in the paper, reusable assets may represent investments of up to several man-years of implementation effort. For most companies, such assets represent a considerable amount of capital, but both engineers and management are not always aware of that. For instance, an increasing number of dependencies (especially implicit dependencies) between assets is a sign of accelerated aging of software and, in effect, decreases the value of the assets. However, since no economic models are available that visualise the effects of quick fixes causing increased dependencies, it is hard to establish the economic losses of these dependencies. In addition, reorganisation of software assets that have been degrading for some while is often not performed because no economic models are available to visualize the return on investment.

The lack of economic models influences several of the identified problems. In general, one can recognize a lack of forces against time-to-market pressure because no business case for sound engineering (versus deadline-driven hacking of software) can presented.

Research issues. One can identify a need for economic models in two situations. Firstly, models are needed for calculating the economic value of an asset, based on the investment (man hours) but also on the value of the asset for future product development and/or for an external market. Secondly, models are needed for visualising the effects of various types of changes and extensions to the asset value. These models could be used to visualise the effects of quick fixes and implicit dependencies on the asset value.

6.5 Encapsulation boundaries and required interfaces

Although many of the issues surrounding product-line architectures are non-technical in nature, there are technical issues as well. The lack of encapsulation boundaries that encapsulate reusable assets and enforce explicitly defined points of access through a narrow interface is a cause of a number of the identified problems. In section 4 we discussed the difference between the academic and the industrial view of reusable assets. Some of the assets at the interviewed companies are large object-oriented frameworks with a complex internal structure. The traditional approach is to distinguish between interface classes and internal classes. The problem is that this approach lacks support from the programming language, requiring software engineers to adhere to conventions and policies. In practice, especially under strong time-to-market pressure, software engineers will go beyond the defined interface of assets, creating dependencies between assets that may easily break when the internal implementation of assets is changed. In addition, these dependencies tend to be undocumented or only minimally documented.

A related problem is the lack of required interfaces. Interface models generally describe the interface provided by a component, but not the interfaces it requires from other components for its correct operation. Since dependencies between components can be viewed as instances of bindings between required and provided interfaces, one can conclude that it is hard to visualize dependencies if the necessary elements are missing.

The lack of encapsulation boundaries and required interfaces primarily causes problems related to component dependencies. For instance, component decomposition is complicated since the new part-components can continue to refer to each other without explicit visibility.

As mentioned, companies address these issues by establishing conventions and policies, but these tend to be broken in practice. Documentation of the assets and inspection of design documents, the implementation and the documentation of assets helps enforce the conventions and policies.

Research issues. The primary research issue to address this cause is to find approaches to encapsulation boundaries that are more open than the black-box component models, but provide protection for the private entities that are part of the assets. Also, more research on the specification and semantics of required interfaces is needed. One example of an existing model is described in [Batory & O'Malley 92]. A second example is the layered object model where an "acquaintance-based" approach is presented that allows for specifying required interfaces and binding these interfaces to other components [Bosch 98b].

7. RELATED WORK

Tools, techniques, and approaches to the development of families of software products have been proposed by a number of authors. LIL [Goguen 86] is an example of a module interconnection language (MIL) that describes component (or module) based systems. In [Neighbors 89], the Draco approach is discussed that, although not using the same terminology, identifies the basic structures of software development based on reuse of domain designs and implementations. [Perry 89] discussed the Inscape environment, focusing on software evolution and problems of scale (i.e., complexity, programming-in-the-large, and programming-in-the-many). [Goguen 96] discusses parameterized programming to instantiate generic descriptions with domain-specific components. [Batory & O'Malley 92, Batory & Geraci 97] discuss a hierarchical component-based model that facilitates the development of families of systems. [Biggerstaff 94] discusses a basic problem in component-based software development, i.e., scaling, and identifies some of the problems discussed in this paper.

With respect to product-line architectures, a number of authors have studied their industrial use. [Macala et al. 96] discuss a demonstration project using product-line development in Boeing in cooperation with the US Navy as part of the STARS initiative. The authors identify four elements of product-line development, i.e., process-driven, domain-specific, technology support, and architecture-centric. The lessons learned during the project are discussed and a set of recommendations is presented. [Dikel et al. 97] discuss lessons learned from using a product-line architecture in Nortel and present six principles: focusing on simplification, adapting to future needs, establishing architectural rhythm, partnering with stakeholders, maintaining vision, and managing risks and opportunities. The report from the product-line practice workshop held by the SEI [SEI 97] presents an overview of the state-of-practice in a number of large software development organisations. Contextual, technology, organizational and business aspects are discussed and a number of critical factors are identified, including deep domain expertise, well-defined architecture, distinct architect, solid business case, management commitment and support and domain engineering unit.

An interesting difference between the papers mentioned above and the results of our study is the perceived necessity of separate domain engineering units. The organisations of our case study explicitly decided against separate domain engineering units. Also [Simos 97] reacts against using domain engineering units and suggests a unified lifecycle model.

[Jacobson et al. 97] presents a complete approach to institutionalizing software reuse in an organisational context, including technology, process, and business aspects. The book is based primarily on experiences from the HP and Ericsson context and contains excellent suggestions also applicable to the interviewed companies.

The Taligent frameworks [Taligent 95] provide various interfaces to each framework, including a client API and a customization API. However, no approaches to language support for high-level encapsulation boundaries are presented. [Szyperski 97] presents an overview of component-oriented programming and discusses the necessity of "required interfaces" in addition to the "provided interfaces". He recognises the necessity of required interfaces, but concludes that current commercial component models focus on provided interfaces only.

8. CONCLUSIONS

The notion of product-line architectures has received attention especially in industry since it provides a means to exploit the commonalities between related products and thereby reduce development cost and increase quality.

In this paper, we have presented a case study involving two Swedish companies, Axis Communications AB and Securitas Larm AB, that use product-line architectures in their product development. Key persons in these organisations have been interviewed and information has been collected from documents and other sources. The goal of the case study was to examine the use, evolution, composition and reuse of assets in a product-line architecture.

In the previous sections, a number of problems and underlying causes are described that were identified in the case study organisations and generalised to a wider context. We have identified three categories of problems related to reusable assets:

1. the existence of multiple versions of assets
2. dependencies between assets
3. the use of assets in new contexts

In the analysis we focus on the causes that we believe underlie the identified problems. The identified causes include

- the early intertwining of functionality
- the organizational structure
- the time-to-market pressure
- the lack of economic models
- the lack of explicit encapsulation boundaries and required interfaces.

In conclusion, product-line architectures can be, and are being, successfully applied in small- and medium-sized enterprises. The studied organisations are struggling with a number of difficult problems and challenging issues, but the general consensus is that a product-line architecture approach is beneficial, if not crucial, for the continued success of these organisations.

ACKNOWLEDGEMENTS

The author would like to thank the software architects and engineers and technical managers at Axis Communications AB and Securitas Larm AB, in particular Torbjörn Söderberg and Rutger Pålsson. Thanks also to the anonymous reviewers for their comments.

REFERENCES

[Batory & Geraci 97] D. Batory and B.J. Geraci, 'Validating Component Compositions and Subjectivity in GenVoca Generators', *IEEE Transactions on Software Engineering*, February 1997, 67-82.

[Batory & O'Malley 92] D. Batory and S. O'Malley, 'The Design and Implementation of Hierarchical Software Systems with Reusable Components', *ACM Transactions on Software Engineering and Methodology*, October 1992.

[Biggerstaff 94] T. Biggerstaff, 'The Library Scaling Problem and the Limits of Concrete Component Reuse', *Third International Conference on Software Reuse*, Rio de Janeiro, November 1-4, 1994, 102-110.

[Bosch 98a] J. Bosch, 'Design Patterns as Language Constructs,' *Journal of Object-Oriented Programming*, Vol. 11, No. 2, pp. 18-32, May 1998.

[Bosch 98b] J. Bosch, 'Object Acquaintance Selection and Binding,' accepted for publication in *Theory and Practice of Object Systems*, February 1998.

[Bosch 98c] J. Bosch, 'Product-Line Architectures in Industry: A Case Study,' *submitted,* June 1998.

[Buschmann et al. 96] F. Buschmann, C. Jäkel, R. Meunier, H. Rohnert, M.Stahl, *Pattern-Oriented Software Architecture - A System of Patterns*, John Wiley & Sons, 1996.

[Dikel et al. 97] D. Dikel, D. Kane, S. Ornburn, W. Loftus, J. Wilson, 'Applying Software Product-Line Architecture,' *IEEE Computer*, pp. 49-55, August 1997.

[Gamma et al. 94] E. Gamma, R. Helm, R. Johnson, J.O. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[Goguen 86] J. Goguen, 'Reusing and Interconnecting Software Components', *IEEE Computer*, February 1986.

[Goguen 96] J. Goguen, 'Parameterized Programming and Software Architecture', *4th International Conference on Software Reuse*, Orlando, Florida, April 1996.

[Jacobson et al. 97] I. Jacobson, M. Griss, P. Jönsson, *Software Reuse - Architecture, Process and Organization for Business Success*, Addison-Wesley, 1997.

[Johnson & Foote 88] R. Johnson, B. Foote, 'Designing Reusable Classes,' *Journal of Object-Oriented Programming*, Vol. 1 (2), pp. 22-25, 1988.

[Kiczales et al. 97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, J. Irwin, 'Aspect-Oriented Programming,' *Proceedings of ECOOP'97 (invited paper)*, pp. 220-242, LNCS 1241, 1997.

[Kruchten 95] P.B. Kruchten, 'The 4+1 View Model of Architecture,' *IEEE Software*, pp. 42-50, November 1995.

[Macala et al. 96] R.R. Macala, L.D. Stuckey, D.C. Gross, 'Managing Domain-Specific Product-Line Development,' *IEEE Software*, pp. 57-67, 1996.

[Neighbors 89] J. Neighbors, 'Draco: A Method for Engineering Reusable Software Components', in T.J. Biggerstaff and A. Perlis, eds., *Software Reusability*, Addison-Wesley/ACM Press, 1989.

[Perry 89] D. Perry, 'The Inscape Environment', *Proceedings ICSE 1989,* 2-12.

[SEI 97] L. Bass, P. Clements, S. Cohen, L. Northrop, J. Withey, 'Product Line Practice Workshop Report,' *Technical Report CMU/SEI-97-TR-003*, Software Engineering Institute, June 1997.

[Simos 97] M.A. Simos, 'Lateral Domains: Beyond Product-Line Thinking,' *Proceedings Workshop on Institutionalizing Software Reuse (WISR-8)*, 1997.

[Szyperski 97] C. Szyperski, *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley, 1997.

[Taligent 95] Taligent, *The Power of Frameworks*, Addison-Wesley, 1995.

Flexibility of the ComBAD* Architecture

N.H. Lassing, D.B.B. Rijsenbrij and J.C. van Vliet
Vrije Universiteit, Faculty of Sciences
De Boelelaan 1081a, 1081 HV, Amsterdam, The Netherlands
tel: +31 (0)20 44 47769, fax: +31 (0)20 44 47653
e-mail: {nlassing, daan, hans}@cs.vu.nl

Key words: Software architecture, software frameworks, software quality, quality assessment, flexibility, adaptability, portability, reusability

Abstract: Software architecture is nowadays regarded as the first step to achieving software quality. The architect's main task is to translate quality requirements into a software architecture. An important step is to assess whether the architecture actually satisfies these quality requirements. The purpose of this paper is to explore which architectural choices support flexibility and how flexibility can be assessed. To that end, we explored the ComBAD architecture, whose main objective is flexibility. We investigated the architectural choices made and assessed whether flexibility was achieved. This will not only increase our insight into flexibility in general, but particularly into the assessment of this quality attribute. We use the term flexibility in the broadest sense of the word: to denote adaptability, portability and reusability. Adaptability can be regarded as flexibility in the narrow sense, portability as the flexibility to use a system in various technical environments, and reusability as the flexibility to reuse part of a system in another system.

1. INTRODUCTION

Recently, there has been an increasing interest in software architecture. It is nowadays generally accepted that the software architecture has a major impact on the quality of an information system. An important step to achieving the desired level of quality is to evaluate the architecture. The

* The ComBAD architecture is a software architecture developed within Cap Gemini Netherlands. ComBAD stands for Component Based Application Development.

software architecture analysis method (SAAM) (Bass *et al,* 1998) was developed with this in mind. SAAM is a scenario-based assessment method that is mainly used to compare the usability of two or more candidate architectures. We claim that SAAM could just as well be used to assess the quality of a single architecture, to evaluate its usability in a certain situation.

The purpose of this paper is to explore how the three elements of flexibility could be addressed in an architecture and how SAAM could be used to assess to what extent the desired level of quality for these attributes was achieved. To do so, we investigated the ComBAD architecture (whose main objective is flexibility in the broadest sense of the word) we described the architectural choices made, and we assessed its adaptability, portability and reusability using SAAM.

The remainder of this paper is organized into four sections. In section 2 we discuss the ComBAD architecture, in section 3 we assess its quality and in section 4 we make some concluding remarks.

2. THE COMBAD ARCHITECTURE

The ComBAD architecture was developed within Cap Gemini Netherlands, a large software house. The architecture originates from a project called "Reuse", whose main purpose was to explore the possibility of reusing domain-knowledge. This project delivered an approach for Component Based Application Development, named ComBAD, and a supporting architecture, the ComBAD architecture, whose main quality requirements were adaptability, portability and reusability. This paper will focus on the ComBAD architecture. The corresponding development approach will not be discussed.

The ComBAD architecture was not developed for a specific customer; it was intended for a broad category of administrative systems. Though it may be used for other domains as well, it is probably more suitable in some situations than others. We return to this point in the evaluation in section 3.

In the next sections, we give an overview of the ComBAD architecture and the architectural choices made to achieve the quality requirements. This overview consists of two parts, a description of the framework, given in section 2.1, and a description of the application architecture, given in section 2.2. These descriptions provide a high-level view of the architecture.

2.1 The ComBAD framework

The quality attributes addressed by the ComBAD framework are portability and reusability. Portability is the quality attribute that indicates the ease with which an application can be moved from one technical environment to another (Delen *et al.*, 1992). In the ComBAD architecture, portability is addressed by using a layered architecture, in which an application is separated from its technical environment, the latter consisting of things like the database-management system used for storage and the protocol used for communications. This separation is achieved by introducing an intermediate layer between the application and its technical environment, which abstracts from the details of this environment. This intermediate layer is the ComBAD framework. The ComBAD framework also offers the type of support required by applications that conform to the ComBAD application architecture.

An instance of this framework is created for a specific development environment and it consists of a number of concrete and abstract classes, and a definition of the way the instances of these classes interact. An application can use a framework in two ways:

1. by inheritance of (abstract) framework classes
2. by calling methods defined in the framework's interface (Lassing *et al*, 1998)

The abstract classes of the ComBAD framework are treated in the next section when we describe the application architecture. In this section, we focus on the interface of the ComBAD framework.

The ComBAD framework provides a common interface to the technical environment by encapsulating access to the environment in a number of services. These services include object brokerage, object persistency, transaction management, notify management, logging and security, each of which is implemented by one component in the framework. The underlying assumption for this is that potential changes in the technical environment each impact just one service and, therefore, also one component. Object persistency, for instance, is implemented by the object-persistency manager, which encapsulates access to the database-management system (DBMS). The impact of changing the DBMS is now limited to this object-persistency manager. Figure 1 shows all of the services of the ComBAD framework. In section 3.3, where we evaluate portability, we assess whether these services encapsulate the environment entirely.

The second quality attribute that the ComBAD framework addresses is reusability. Reusability is much harder to achieve than portability because it is more than a technical problem. Consider the following statement from van Vliet (van Vliet, 1993). He states that "a reusable component is to be valued, not for the trivial reason that it offers relief from implementing the functionality yourself, but for offering a piece of the right domain knowledge, the very functionality you need, gained through much experience and an obsessive desire to find the right abstractions."

Apparently, reuse is only possible within a specific domain and we need a thorough understanding of this domain to determine its reusable elements. We define a domain as a well-defined area of application that is characterized by a set of common notions.

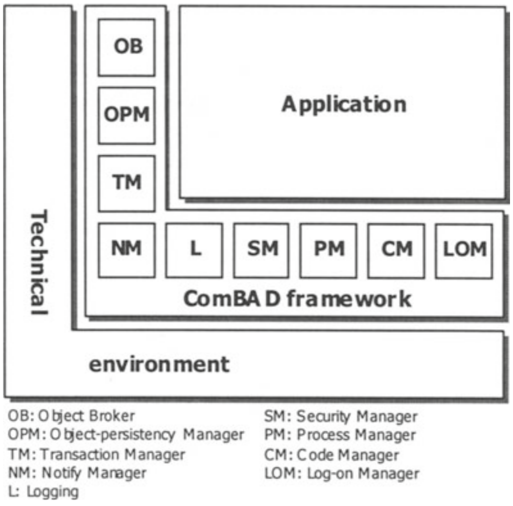


Figure 1. The layered architecture with the ComBAD framework and its services

The ComBAD framework tries to address reusability in two ways. First, the framework itself can be reused. The domain in which this framework could be reused is the technical foundation of applications using the ComBAD architecture. Thus, the reusability of the ComBAD framework depends on the usability of the ComBAD architecture, which is the topic of the evaluation in section 3.

The second way in which the ComBAD framework addresses reusability is that it can serve as an environment for reuse of software components. This addresses one of the technical problems of reuse, namely architectural mismatch. Architectural mismatch occurs when the assumptions that a component and its environment make about each other are conflicting (Garlan *et al.*, 1995). Frameworks reduce the risk of architectural mismatch, because they provide a known environment for components to operate in.

The ComBAD application architecture determines the types of components that can be used in the framework. They are included in the framework as abstract classes that implement the behavior that is common for these components. The actual components of the application are derived from these abstract classes.

Although frameworks address architectural mismatch, they are not a panacea for reuse. They do not relieve the developer from the painstaking process of finding the right components in a domain. However, they do provide support after the right components have been found.

2.2 The ComBAD application architecture

The quality attributes that are addressed by the ComBAD application architecture are adaptability and reusability. According to Bass *et al.* (1998), adaptability is largely a function of locality of change. This means that to increase adaptability, one should try to limit the impact of changes to a small number of components. On the other hand, we should try to limit the number of potential changes by which each component may be impacted.

In the ComBAD application architecture, adaptability is addressed by dividing an application into three layers:

1. the interface layer
2. the processing layer
3. the data layer

The interface layer handles the communication with the environment, consisting of users and other systems. The processing layer contains the application logic. Finally, in the data layer all data of the application is managed.

By separating an application into these layers, changes to the interface of the system can be limited to the interface layer, changes to the application logic limited to the processing layer and changes to the data limited to the data layer. However, this means that the number of potential changes that may impact each component is rather large. Therefore, it was decided to further divide the layers into components, as shown in Figure 2. The interface layer is divided into human interface components (or HICs), which handle a dialog with the user, and system interface components (or SICs) that communicate with other systems. The processing layer is divided into task-management components (or TMCs) that each implement (only) one function. And the data layer is divided into problem-domain components (or PDCs), which record data about a concept from the problem domain. Collectively, these components are called application components.

This division can help us to limit the impact of changes to a few relatively small components. True locality of change is achieved for changes that affect the internals of one or more application components, but leave their interfaces intact. However, some changes not only affect the internals of one or more application components, but also the interfaces of some of them. This means that all dependent application components need to be changed as well. By restricting the dependencies between components, the impact of changes can be restrained. In the application architecture only top-down dependencies are allowed, i.e., an HIC or an SIC should only be dependent on one or more TMCs, a TMC on one or more PDCs and PDCs should be independent of other application components. Notify management is used to inform higher layers of events occurring in the lower layers.

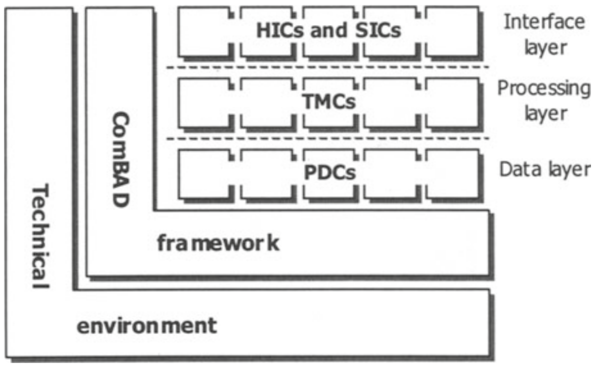


Figure 2. The ComBAD application architecture

Independence also affects reusability, because reusability demands that components are as independent as possible. Independence and, hopefully, reusability of PDCs is increased by prohibiting direct relations between them. This restriction reduces PDCs to stable building blocks that can be reused in other applications. We address the reusability of these components in section 3.4.

The ComBAD framework and the ComBAD application architecture very much depend on each other. First, the components of the application architecture use the services of the framework. For instance, the PDCs are accessed through the object broker and the PDCs use the object-persistency manager for storing themselves in a database. Second, the application components are derived from abstract classes provided by the framework. These abstract classes provide behavior common for each of these types of application components. Thus, the framework and the application architecture cannot be separated; they are highly intertwined.

3. ASSESSING THE QUALITY

To assess the quality of the ComBAD architecture we use the software architecture analysis method, SAAM (Bass *et al.*, 1998). This is a scenario-based method that consists of formulating a number of scenarios and evaluating the impact of each on the architecture. A scenario is a situation that can occur in the life of an architecture. Although SAAM was developed to compare two or more candidate architectures, we use it to assess the quality of a single architecture. The first step in the evaluation is to derive a number of scenarios from the quality requirements of the architecture. For example, from the quality requirement portability we can derive the following scenario: What happens when another DBMS is to be used? By formulating a number of these scenarios, we can make portability tangible, because they capture what we actually want to achieve with portability.

The next step is to evaluate the impact of these scenarios on the architecture. We classify the impact of a scenario into four discrete levels. At the first level, no changes are necessary, which means that the scenario is already supported by the architecture. At the second level, just one component of the architecture needs to be changed, but its interface is unaffected. At this level, we have true locality of change. At the third level more than one component is affected, but no new components are added or existing ones are deleted. This means that the structure of the architecture remains intact. At the fourth level architectural changes are inevitable, because new components are necessary or existing ones become obsolete. It is clear that one should seek to keep the level of impact as low as possible.

When we return to our example scenario, we see that this scenario necessitates a change in the object-persistency manager. Thus, this scenario has a level two impact. This means that we have locality of change for this scenario and that the architecture is portable with respect to the DBMS used.

We have created four categories of scenarios. The first two categories, which focus on adaptability, contain scenarios that are related to the requirements of the system. We have made a distinction between scenarios that address technical adaptability and those that address functional adaptability. The former consists of scenarios that explore the applicability of the architecture in situations with different technical requirements. The latter consists of scenarios that explore the effect of changes in the functional requirements. The third category of scenarios concentrates on portability, which is evaluated by scenarios that simulate changes in the technical environment. The final category focuses on reusability. This category includes scenarios that explore the use of elements of the architecture in other systems and architectures.

3.1 Technical adaptability

Technical adaptability is the flexibility of an application to incorporate changes to the technical requirements. The scenarios simulate the use of the

ComBAD architecture in situations with diverse technical requirements. Note that the ComBAD architecture was not specifically developed for some of these situations. The architecture is usable in a situation when the scenario has an impact of level three or lower. The results of these scenarios are summarized in Table 1.

Scenario 1: *Which changes are needed when the architecture is to be used for secure applications?*

We assume that for secure applications a number of things are necessary. First, each user action should be authenticated and it should be possible to grant different levels of access to users (for example, no access, read-only, full control, etc.). This is already supported by the ComBAD architecture, so it is unaffected. Second, the communication between clients and servers should be encrypted. Encrypted communication is not yet present in the architecture, but it could be added by changing one of the base classes for the application components. Finally, access to servers should be prohibited for unsecured hosts. This means that the log-on manager should be changed so that it inspects the network address of clients. Our conclusion is that using the architecture for secure applications necessitates changes to a number of the existing components and, therefore, this scenario has a level three impact.

Scenario 2: *Which changes are needed when the architecture is to be used for real-time systems?*

The distinguishing features of real-time systems are deadlines and synchronization between different parts of a system (Laplante, 1993). These features are not supported by the ComBAD architecture. However, deadlines could be enforced by introducing something like a deadline manager into the framework, that makes sure that a systems responds within a certain period. Similarly, synchronization could be added by introducing a synchronization manager that makes sure that the different parts of a system operate in harmony. In addition, the division of applications into H/SICs, TMCs and PDCs is perhaps not usable for real-time systems. So, the impact of this scenario is architectural and it is classified as level four.

Scenario 3: *Which changes are needed when the architecture is to be used for ultra-reliable systems?*

In ultra-reliable systems both software and hardware are often replicated (Leveson, 1995). This redundancy makes sure that the system remains in working order after one or more services have failed. In addition, these systems could use voting, which means that the same operation is performed by two or more elements, and the end result of the operation is some kind of weighted average of the results of individual elements. Both redundancy and voting could be addressed by introducing one or more front-end servers that encapsulate the access to the other services. This has a major impact on the architecture and, therefore, the impact of this scenario is classified as level four.

Scenario 4: *Which changes are needed when a Web interface is created for an application?*

To make the system accessible from a Web browser, the human interface components (HICs) should be replaced with applets that can be viewed within a Web browser. Because the lower layers are independent of the HICs, they are unaffected by changes in the HICs. The HICs are the only components affected and therefore the impact of this scenario is classified as level three.

Scenario 5: *Which changes are needed when the architecture is used for a system that uses workflow management?*

The ComBAD framework already has a process manager that controls which operations may be performed by the user in a certain situation. This component could be enhanced to support true workflow management. Since the process manager is the only component affected, the impact of this scenario is level two.

Table 1. Summary of the scenarios for technical adaptability

	ComBAD framework		Application				
Scenario	Archi-tecture	Component s	Archi-tecture	HICs/ SICs	TMCs	PDCs	Imp. level
1	-	M	-	-	-	-	3
2	+	M	+	?	?	?	4
3	+	M	+	?	?	?	4
4	-	-	-	M	-	-	3
5	-	O	-	-	-	-	2

- = unaffected, + = needs to changed, O = one component affected, M = more components affected, ? = undefined

As expected, we see in Table 1 that the architecture is not directly usable in every situation. Using it for real-time or ultra-reliable systems necessitates major changes to the architecture. In the other situations, the architecture is usable, but some changes are necessary. These changes sometimes affect the framework and sometimes the application components. When the ComBAD architecture is used in an actual situation, more scenarios are probably needed to evaluate whether the right services are identified to encapsulate the expected changes to the technical requirements.

3.2 Functional adaptability

Functional adaptability is the ease with which changes in the functional requirements can be implemented. It is difficult to address the functional adaptability of an architecture, due to the absence of functional requirements. However, we are able to address the architectural aspects of changes to the functionality. To this end, we use scenarios that explore the effect of adding or deleting components from the application. The results are summarized in Table 2.

Scenario 1: *Which changes are needed when a problem-domain component (PDC) is added or deleted?*

When a new PDC is added, one or more elements in the higher layers should also be modified, for it does not make any sense to add a PDC without using it in one of the higher layers. When a PDC is deleted, the components in the higher layers that are dependent on it should be changed. The impact of this scenario can therefore be classified as level three.

Scenario 2: *Which changes are needed when a task-management component (TMC) is added or deleted?*

A task-management component is always invoked from the interface layer. Therefore, when a TMC is added, one or more H/SICs need to be changed to make use of this new TMC. Similarly, when a TMC is deleted, one or more H/SICs need to be changed to remove any references to the TMC. This scenario affects one TMC and at least one, but possibly more, H/SICs and the impact of this scenario can therefore be classified as level three.

Scenario 3: *Which changes are needed when a human/system interface component (H/SIC) is added or deleted?*

The impact of this scenario is very small, because no other components are dependent on the H/SICs. In fact, the H/SIC that is added or deleted is the only component that is affected. Thus, this scenario has a level two impact.

Table 2. Summary of the scenarios for functional adaptability

	ComBAD framework		Application				
Scenario	Archi-tecture	Components	Archi-tecture	HICs/ SICs	TMCs	PDCs	Imp. level
1	-	-	-	M	M	O	3
2	-	-	-	M	O	-	3
3	-	-	-	O	-	-	2

- = unaffected, + = needs to changed, O = one component affected, M = more components affected, ? = undefined

From Table 2 we conclude that changes to the functional requirements do not affect the ComBAD framework. This means that the framework is entirely separated from the functionality of the application. And as expected, we observe that TMCs are unaffected by changes to the interface layer and that PDCs are unaffected by changes to either the processing layer or the interface layer.

3.3 Portability

At first sight, portability and technical adaptability very much look alike, but they are not the same. Portability is the ease with which a system can be adapted to changes in the technical environment and technical adaptability is the ease with which a system can be adapted to changes in the technical requirements. The scenarios in this category explore the effect of changes in the technical environment.

Scenario 1: *Which changes are needed when another database is used?*

This scenario was used in the introduction of this section. The object-persistency manager is the only element that is impacted. Thus, the impact of this scenario can be classified as level two.

Scenario 2: *Which changes are needed when another operating system is used for the client machines?*

The answer to this question is not unambiguous, because it depends on the programming language and the development environment used. First, if the application is written in Java, no changes should be needed, but other languages may cause major problems. Second, it is important which development environment is used, because a number of development environments are able to generate and/or compile code for different platforms. This approach is taken in ComBAD, where the tools used can generate and/or compile code for multiple platforms. This solution lies outside the architecture, and the impact of this scenario can be classified as level one.

Table 3. Summary of the scenarios for portability

	ComBAD framework		Application				
Scenario	Archi-tecture	Component	Archi-tecture	HICs/ SICs	TMCs	PDCs	Imp. level
1	-	O	-	-	-	-	2
2	-	-	-	-	-	-	1

- = unaffected, + = needs to changed, O = one component affected, M = more components affected, ? = undefined

In Table 3, we observe that changes in the technical environment affect very few components of the ComBAD architecture. We notice that the application components are unaffected by our scenarios, which could indicate that the framework actually encapsulates access to the environment. However, there may be potential changes in the technical environment, not mentioned here, that have an impact above level two.

3.4 Reusability

We have chosen to assess reusability by scenarios that test the usability of ComBAD components in other situations, as well as the usability of other components within the ComBAD architecture. These scenarios focus on individual components, so it is not very meaningful to create a table that indicates which elements of the architecture are affected.

Scenario 1: *Can components that were not especially developed for the ComBAD framework be used in applications built using the ComBAD architecture?*

The components that can be reused in these applications are mainly GUI-controls, like ActiveX-controls and Java Beans. However, because of the demands these components put on their environment, using them limits the portability of an application. Other components could be reused in these applications as well, if the components on which they depend are also included.

Scenario 2: *Can the application components be used in systems using another application architecture?*

The application components are usable in an environment that provides all of the framework services used by the component. This means that, theoretically, application components are reusable in another application, but it will require an enormous amount of work if they depend on more than a few framework services.

Scenario 3: *Can the object broker of the ComBAD framework be reused in systems using another architecture?*

The answer to this question is yes, provided all of the components upon which the object broker depends, being the transaction manager, the notify manager and the object-persistency manager, are included in the other architecture as well. However, this answer focuses on the architectural aspects only. Whether the object broker offers the right functionality in this situation is ignored.

Scenario 4: *Can application components be reused in other applications using the ComBAD architecture?*

Architecturally speaking, application components can be reused in other applications using the ComBAD architecture, provided the components on which they depend are also included. However, the reusability of a component also depends on whether it offers the right functionality. Within the ComBAD project, it was felt that the level of abstraction of the application components is too low. Therefore, a number of these components are grouped into packages, the same way Jacobson *et al.* (1997) address reusability. Whether these packages offer the right functionality can only be judged in an actual situation.

From these scenarios, we conclude that it is hard to assess the reusability of components, because it largely depends on the functionality they implement. From an architectural point of view, we may conclude that most components of the ComBAD architecture could be reused, but that this is easiest within the ComBAD architecture.

3.5 Evaluation of the assessment

In this section, we assessed the flexibility of the ComBAD architecture using scenarios. The assessment showed that the technical adaptability and portability of a single architecture could be assessed quite well using scenarios, yet functional adaptability and reusability are harder to assess. The main difficulty of the assessment of functional adaptability of architectures is that functional requirements are lacking, which means we can only address the architectural aspects of changes to the functionality.

Reusability is hard to assess in general, because the reusability of a component largely depends on whether it supports the right functionality, which can only be judged by a developer.

In addition, the assessment demonstrated that the flexibility of an architecture should always be related to the area of application. Although the assessment given in this section provides some general insight into the usability of the ComBAD architecture, one is unable to value the scenarios but in an actual situation.

4. CONCLUSION

The purpose of this paper is to explore how flexibility can be addressed in an architecture and how we can assess whether an architecture supports it. To that purpose we have examined the ComBAD architecture. In the first part of this paper we presented the architectural solution, which consists of the architectural choices made to address the quality requirements: adaptability, portability and reusability. We showed that in the ComBAD architecture portability and reusability are addressed by creating the ComBAD framework and that adaptability and, once again, reusability are addressed by the application architecture.

In the second part of this paper we assessed the flexibility of the ComBAD architecture. To do so, we formulated scenarios for assessing technical adaptability, functional adaptability, portability and reusability. It turned out that assessment using scenarios of technical adaptability and portability of a single architecture is quite possible. However, functional adaptability and reusability proved to be hard to assess using scenarios, because we are considering an architecture, lacking functional requirements and actual application components.

The assessment demonstrated that the introduction of the ComBAD framework encapsulates changes to the technical environment from the application. In addition, we showed that the framework is unaffected by changes in the functional requirements. However, whether the services in the framework encapsulate the right technical mechanisms could be a topic for further research. In addition, one should always remember that flexibility is a relative notion, which can only be valued in a particular context.

Our next step will be to investigate the architecture of an existing system to see whether we are able to assess its quality attributes, including functional adaptability and reusability of components. This way we hope to deepen our insight into the quality attributes and their assessment in software architectures.

ACKNOWLEDGEMENTS

This research is mainly financed by Cap Gemini Netherlands. We are grateful to Cor de Groot, Ad Strack van Schijndel, and Guus van der Stap of Cap Gemini Netherlands for their time and comments. Guus van der Stap (e-mail: GStap@inetgate.capgemini.nl) can be contacted for more information about ComBAD.

REFERENCES

Bass, Len, Paul Clements and Rick Kazman (1998). *Software architecture in practice.* Addison Wesley, Reading.

Delen, G.P.A.J. and D.B.B. Rijsenbrij (1992) The specification, engineering and measurement of information systems quality. *J. Systems Software* **17**, 205-217.

Garlan, David, Robert Allen and John Ockerbloom (1995) Architectural mismatch: Why reuse is so hard. *IEEE Software* **12**, 6, 17-26.

Jacobson, Ivar, Martin Griss and Patrik Jonsson (1997). *Software reuse: architecture, process and organization for business success.* ACM Press, New York.

Johnson, Ralph E. (1997) Frameworks = (Components + Patterns). *Communications of the ACM* **40**, 10, 39-42.

Laplante, Phillip A. (1993) *Real-time systems design and analysis: an engineer's handbook.* IEEE Press, New York.

Lassing, N.H., D.B.B. Rijsenbrij and J.C. van Vliet (1998) A view on components. *Proceedings of the 9th International DEXA Workshop on Database and Expert Systems Application.* IEEE Computer Society, Los Alamitos, 768-777.

Leveson, Nancy (1995) *Safeware: system safety and computers.* Addision Wesley, Reading.

Shaw, Mary, and David Garlan (1996) *Software architecture: perspectives on an emerging discipline.* Prentice-Hall, Upper Saddle River.

Van Vliet, Hans (1993) *Software Engineering: principles and practice.* John Wiley & Sons, Chichester.

Medical Product Line Architectures
12 years of experience

B.J. Pronk
Philips Medical Systems
P.O. Box 10,000, 5600 DA Best, The Netherlands
bpronk@best.ms.philips.com

Key words: Example architectures, product line architectures, styles and patterns

Abstract: The product line architectures for diagnostic imaging equipment like CT, MRI and conventional X-Ray have to cope with large variations (in hardware and application functions) combined with a high level of integration between their embedded applications. Therefore a primary goal of these architectures is to avoid monolithic applications while retaining the required integrated behaviour. Furthermore, an easy and independent variation of the constituting components is essential. The product line architecture described in this paper gives one recent example solution to this problem. This example presents a layered, event-driven, resource-restricted system based on the model-view-controller pattern. Its technical implementation relies heavily on state of the art desktop (Windows NT™) and component techniques (DCOM). For this architecture, orthogonality and (binary) variation have been the key design goals. Three views of this architecture—the conceptual, technical, and process models—are discussed. In all three views the rationale of the chosen concepts and their relation to the problems indicated above is shown.

1. MEDICAL ARCHITECTURES

Philips Medical Systems is one of the world's leading suppliers of diagnostic imaging equipment. Its product range includes conventional X-ray, computed tomography (CT), magnetic resonance imaging (MRI), and ultrasound (US) equipment. These product families, usually called modalities, come in many variants of which only small quantities (100-1000) are being produced, enforcing reuse of development effort and product family architectures for all of them. In this paper the main issues encountered in the architecture development of these product families will be discussed. For illustration a recent example architecture will be presented.

1.1 Characteristics of medical software environment

The main characteristics of the Philips Medical embedded software development environment are:

- distributed, multi-processor
- real-time embedded and standard desktop environments
- large amount of code (> 10^6 lines of code) per system
- large software engineering groups (> 100 FTE's)
- software is by far the fastest growing component of all products
- long product support, maintenance and extensions (10-15 year)
- Long-running projects (2-3 years)
- distributed development
- small product series (< 1000/year)
- strict quality, legal, and safety requirements

1.2 Architecture overview

From a physical viewpoint most of the products mentioned above are constructed along the same principles. They are centred around a host processor, running a desktop operating system, that controls a set of modality-specific peripheral devices that are needed to generate, process, and view images. These peripherals are normally large, expensive, and controlled locally by embedded real-time processors or digital signal processors. Examples are high-tension amplifiers, patient support mechanics, RF-coils, etc. The set of peripherals is unique to a single product family, although many variations of individual peripherals are usually supported within one product family.

On the host of all modalities, similar software applications linked with the user workflow can be identified:

- database and patient administration, for entering patient data in the system
- acquisition, which programs all devices for image generation,
- a viewing application that allows the user to review and process the acquired images,
- image handling applications that support all further handling of the information obtained during the examination, such as printing, archiving, and network communication.

The architecture is outlined in *Figure 1.*

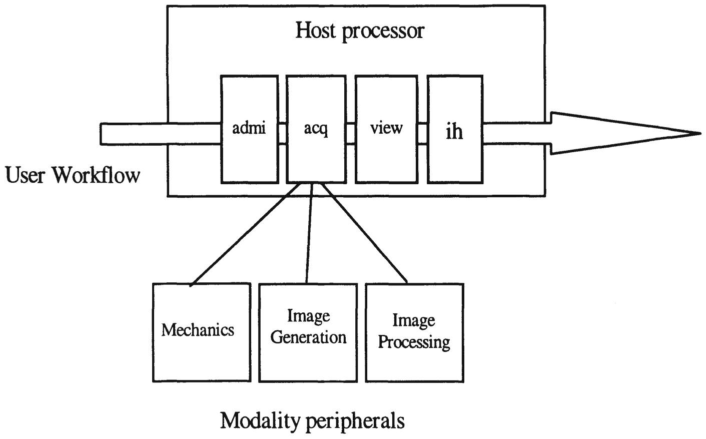


Figure 1. Medical architecture overview

2. MAIN ARCHITECTURAL ISSUES

The main issues to be addressed by the software architecture of medical product families can be summarised as:

- Reuse: The need to support many different product family members, serving a variety of application areas and operating in many different (hardware) configurations, with one shared code base.
- Independence: Allow parallel, independent, and incremental development for specific family members.
- Time to market: Allow efficient addition of new functionality for the various family members in reaction to changing market needs.

In the remainder of this section the main aspects of these problem areas are explored somewhat further.

Reuse:

Medical products come in many configurations (types of hardware, software options) serving various market segments and application areas. Yet within one product family (X-Ray, MRI etc.) a lot of functionality can be identified that is common to all family members. Because of the long lifetime, small production numbers, and enormous code base (investments) of most product families these variations must be handled by the configuration of a single basic platform.

Independence:

Often the variations indicated above influence significant properties of the system (e.g., maximum frame speed), that propagate throughout the entire architecture. As a consequence of this, current implementations show cross-dependencies throughout the entire software system. Other symptoms of these phenomena are multiple definitions and extensive and complex branches.

Furthermore the current practice of source code reuse introduces heavy compile-time dependencies between all components. Independent development and delivery are virtually impossible in this situation. Furthermore this strong coupling requires extensive testing at every change, yielding ever-longer test cycles.

The software applications of a medical device present very integrated behaviour to their users. This is reflected in software dependencies at all levels (user interface, application and technical level). Examples of these are the sharing of the current patient and image between applications, the use of shared (hardware) resources, and the compensation of imperfections of one device in another one.

Time to market:

Many new features, acquisition techniques, and hardware devices are added to medical products over the lifetime of the software architecture. These extensions are often accompanied by extensive growth of coupling in the system, since the necessary interfaces do not exist in the architecture. Continuous engineering by an ever-varying population of developers, forgetting or even unaware of the original architecture, further aggravates this situation.

Medical devices contain a lot of persistent data: patient and image related data, system settings, and configuration of the system and its components and calibration data. Each of these settings depends on the software level of the components, the actual available hardware, and the configuration and options available on a system. This strongly coupled set of data imposes a significant barrier to change. The same goes for exchange of data between different releases, systems, and off-line tools that introduce many compatibility problems.

Dedicated solutions and proprietary techniques have been widespread throughout the professional industry. In view of the advance of modern desktop operating systems with their myriad applications, productivity tools, and high innovation rate, this legacy has become one of the sources of a low rate of innovation in the industry.

3. AN EXAMPLE SOLUTION

In this section a recent example of medical product family architecture is described. In its quest for a solution to the three main architectural issues introduced in the previous section, the architecture applies the following principles.

1. avoiding a monolithic design by de-coupling and localisation. Every component can be replaced in isolation.
2. binary reuse of components, reducing compile time dependencies.
3. use of standard technology and tools for productivity enhancement
4. division of the product family development into a generic (platform) part and member-specific parts. Addition of specific parts should be possible in independent parallel activities.

None of the principles stated above is very revolutionary, and of them only binary reuse of components can be considered to be relatively new, since enabling technology has recently become widely available (COM, CORBA). Yet we think that the strict adherence to these principles and the actual implementation followed has led to a system coping with the main architecture issues better than any of our previous implementations.

This new product family architecture has been modelled in several views, which will be described in detail in the remainder of this paper:

- the conceptual architecture view: Describing the solutions and rules as applied to tackling the main architectural issues of decomposition vs. co-operation. The actual design of the system employs these solutions. This view will receive most attention in the discussion in this paper.
- the technical architecture view: This view describes all additional constructs necessary on top of the conceptual view (e.g., I/O classes, caching mechanisms) to realise the system. It also describes the hardware (processors, buses etc.) and software (operating system, protocols etc.) infrastructure and technology choices.

We will also describe the process architecture. However this is not viewed on the same level as the previous two; in fact within both the technical and conceptual architecture a process architecture view can be identified. Within the conceptual architecture this describes the general approach for handling the required (application) concurrency. Within the technical architecture it describes the deployment of the elements of the decomposition into threads, processors, and processes. This latter point will not be addressed in this paper.

The architecture is thus described by

- a set of rules and concepts,
- a series of technology and infrastructure choices,

- the decomposition of the solution domain into so-called Units,
- their deployment to the infrastructure, and
- the set of interfaces between them.

As much as possible the rules and concepts are expressed in formal terms, to allow automatic verification of adherence to them in both the platform and specific developments.

The presented three models (conceptual, technical, process) closely resemble three of the views described by Kruchten (Kruchten, 1995). On a lower level the same views are used in the design of the individual Units that fit into this architecture. This set of views has been selected since they have proven to be sufficient input for the designers of these Units to complete their requirements and designs in relative independence.

3.1 Conceptual architecture

The conceptual architecture of the product family describes the concepts, rules, and tactics that implement the principles described above. Note that the conceptual architecture mainly addresses principle 1 (localisation and de-coupling). Note also that the conceptual architecture is almost independent of the underlying technology, which is added only at a later stage.

3.1.1 Layering

The product family architecture decomposes the system into a number of (independent) abstraction layers, from the bottom up, as shown below in *Figure 2*.

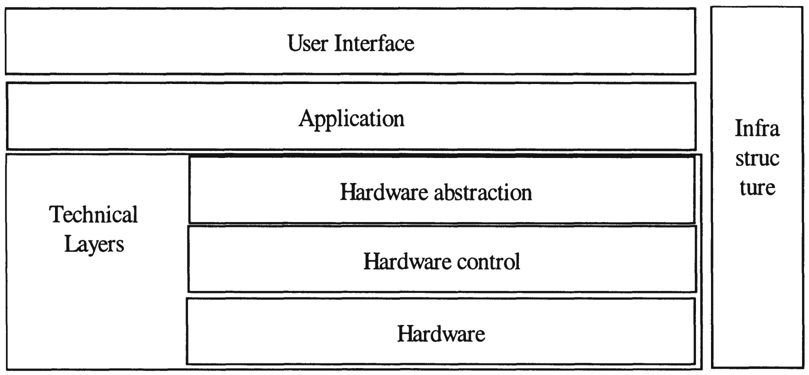


Figure 2. Layered set-up of product family architecture

The layers are

- Technical layer, consisting of the following sub layers:
  - Hardware: basic digital and analogue hardware and their controllers.
  - Hardware control: drivers and real-time control of hardware that shield the low-level details of the hardware implementation such as registers, addresses, interrupts, etc.
  - Hardware abstraction: an abstraction layer offering a domain-specific abstraction of the underlying type of hardware (e.g., the X-Ray generation part in a CT-family).
- Application layer: The actual user functions realised with this equipment.
- User interface layer: The presentation layer, taking care of display and user interaction.

Next to these three layers there is an infrastructure layer that is used by all. The three layers and their sub-layers supply a true abstraction, i.e. they are not transparent to the layers above them. Each of these layers can therefore be replaced independently of the surrounding layers. This is one of the major features supporting variation within the product family. Examples of this are:

- different user interfaces for the members of the family
- various implementations of the geometry part of an X-Ray system
- implementing functions from several application areas on top of the common (domain) abstraction layer (e.g., a cardiological and a neurological MRI application)

3.1.2 Conceptual building blocks

Within each layer several independent Units are distinguished, which should not interact with each other. Therefore each of these Units is as self-contained as possible. The conceptual building blocks used within the three layers are:

- Services: The service concept is a main structuring element of the architecture. A service is a software entity that autonomously executes a number of tasks for another part of the software, guarding a set of resources. A service is a completely isolated part of the architecture that also keeps its own configuration, etc. The technical layer consists of a set of these services, one for each device. In an X-ray system, for example, the services are for the generator and detector, and in an MRI system the services are for the gradient amplifier and the RF coils.
- Applications: There are a number of applications such as reviewing, acquisition, patient, and beam positioning, etc. These applications are also services offering an interface to the user interface layer. Applications

offer a very uniform interface consisting of commands (in fact the use cases as defined in the functional specifications) and a so-called UI model that represents all information (data and state) necessary for the user interface.

- User interfaces: The user interface is completely decoupled from the applications and interacts with them through the application service interfaces described above. Throughout the system a model-view-controller pattern is applied, with the user interface being the "view." The application in fact contains the model (the UI model) and the controller (the commands). The grouping and appearance of the user interface is not known by the applications. There might be one integrated UI for multiple applications or a single user interface per application.

3.1.3 Independence

The previous steps represent a major step forward in decoupling the various Units of the system. However interaction between Units cannot be avoided completely because of the integrated behaviour aspects describedabove. Yet we maintain the rule that applications and services of the system will not interact directly with each other. This will be supported by the following mechanisms.

- Event driven: Another main concept of the architecture that supports decoupling is notification. Objects in this architecture may issue events (notifications) that can be received (if requested) by so called observers. This mechanism works both within processes and across process boundaries. The source of the events in this mechanism is not aware of its observers. All upward communication between Units is based upon this mechanism.
- Integration: All system-wide known data (e.g., patient data, but also currencies) is stored in a separate service called the integration and data model. Applications never directly exchange data such as a change in the current patient. Instead the current patient object in the integration service is updated by the patient administration application. All interested applications may be notified of this change through the notification mechanism just described. The integration service is closely linked to the database since a lot of this information is also stored persistently.
- Automation: Many sequences of operations in the system are pre-programmed. After an acquisition, the system switches to reviewing mode: data are forwarded to an archive etc. After closing an examination, data are forwarded to printers, the Radiology Information System, etc. Such functionality is located in a separate automation service that

receives completion notifications from the applications and starts the relevant actions. Again, the applications do not interact directly.

- All system characteristics are derived from the available resources an application can obtain from the services it uses. There is no hard coding in the applications of restrictions of the underlying services. This implies that resources may be added to increase the capabilities of the system without additional coding.

3.2 Process architecture

This section describes the concepts used for decomposing the system into separate concurrent processes, starting with the application requirements for concurrency. From a user point of view the system should deliver the following levels of concurrency.

- Multiple users operating separate applications concurrently. This will be handled by defining all applications to be separate processes.
- All long-running, non-interactive user functions (e.g., printing, export, archiving) have to be performed as background parallel processes since the user wants to be free to do other actions while these functions are executed.
- Long-running interactive user functions (like screen build up) have to be performed in parallel processes to retain user interface responsiveness. For these functions the user should be able to cancel or overrule it.

From a technical point of view additional concurrency is introduced in the system since asynchronous hardware has to be controlled. So all services handling hardware have to be separate conceptual processes. Yet another technical point of concurrency follows from the services concept itself. Lengthy actions are often distributed over a client and multiple services. A service request may take considerable time to complete since the handling of hardware I/O often is involved. During the time that the service request is handled, the application can often do other useful things (e.g., starting other service requests in parallel). It is a matter of choice where to put the conceptual processes for handling lengthy service requests. We choose to put them in the services themselves. So, all lengthy service requests have to run in separate conceptual processes. This also implies that these service requests will complete asynchronously (and use notification to signal completion).

From this initial selection even more concurrency requirements can be derived. Since multiple conceptual processes are active in parallel, shared resources (e.g., database, context) are introduced. Therefore additional conceptual processes will be introduced to serialise access to these shared resources.

3.3 Technical architecture

The technical architecture of the system supports, in particular, the following principles from the introduction to section 3:
- binary reuse of components
- enhanced productivity by application of standard, state-of-the-art technology
- building a generic platform with product-specific additions

3.3.1 Use of a standard environment

Professional industrial environments have long worked with proprietary solutions. However the advance of standard desktop environments, market pressure, and the need for productivity increases drive the industry towards usage of standard solutions and open standards. Note that this is not only a matter of money. Even where money is no argument, the time and people needed to create from scratch something to compete with standard desktop environments represents a tremendous bottleneck. Finally, the innovation rate of desktop environments is now so high that proprietary solutions will probably be outdated before they are introduced. Therefore the following approach is chosen for the new product family architecture:
- Allocate, as much as possible, software functionality to custom hardware components. Only build dedicated hardware when processing/responsiveness cannot (cost effectively) be delivered by such a platform.
- Allocate, as much as possible, functionality in a standard desktop environment. Only use a real-time operating system environment when strictly required (for performance, safety, or graceful degradation).
- Use standard PC-architecture and technology as much as possible (PCI, Intel x86, Windows NT, Microsoft Foundation Classes, Windows User interface, Windows NT services etc.).
- Use standard software packages (database, license management, network)
- Use internet technology (Java/HTML/Browser, Windows NT peer web server) for (remote) service.

3.3.2 Binary exchange

Classical reuse programs are often based on source code level reuse. This approach introduces strong compile-time dependencies. Furthermore it does not support true reuse, since extensive testing is still required in the new code/compile environment. This situation is even further aggravated when using object-oriented languages and deep inheritance trees. Based on these experiences it has been decided that the new product family architecture will be based on binary variation. The following choices have been made in this area:

- Component-based development (DCOM) based on binary exchange, allowing flexible allocation of UI, application and services.
- All interfaces in the system will be expressed in IDL, and DCOM will be used for all communication between Units.
- All notification between Units will be based on the COM connection point mechanism.
- DCOM, however, is only used as an interface mechanism; all implementation classes are strictly separated from this interface shell.
- Interfaces are considered immutable even when extending, for example, ranges of enumerated types or error codes; new interface versions will be introduced.
- Apply component technology to define frameworks for all extensible parts of the system. A framework consists of a set of interfaces and some generic functionality. For example, the acquisition application is a framework in which (binary) components can be added to support additional acquisition procedures.

4. CONCLUSIONS

Medical equipment architecture has to focus on orthogonality and independence to support a viable product family concept. The rigorously pursued decoupling in the presented product family architecture allows for the development of completely localised and highly independent components. The use of DCOM as standard interface technology enables versioning, strict interface management, and the delivery of components that are thoroughly tested. In addition, applying standard technology and components will reduce time to market significantly. This combined approach has resulted in a generic platform, which through addition of system-specific components, can be specialised in parallel developments.

REFERENCES

Kruchten, Philippe B. (1995) The 4+1 View Model of Architecture, *IEEE Software*, November 1995, pp. 42-50.

Kaleidoscope

A Reference Architecture for Monitoring and Control Systems

Andrea Savigni, Francesco Tisato
DSI Università di Milano
Via Comelico 39/41 20153 Milano, Italy
emails: {savigni|tisato}@dsi.unimi.it
voice: +39 2 55006231
fax: +39 2 55006249

Key words: Software architecture, monitoring and control systems, connectors, strategy, object orientation

Abstract: Although monitoring and control systems can be applied to a great variety of application domains, they exhibit a number of common characteristics, particularly the extensive use of abstraction layers and information streams. This paper presents a reference architecture upon which a number of monitoring and control systems for a wide range of application domains can be designed. The architecture is described in terms of components and connectors, and the UML methodology is employed to specify class diagrams. The architecture is specifically conceived to be made of reusable components; to that aim, a clear separation is made between information components and strategic components, so that the former can be reused under different strategies. *Conceptual images* are information components that model concepts of the application domain, and are specialised in terms of *concrete images*, such as acquisition, processing, and presentation. The major task of the system is to align concrete images, which takes place via transfer of objects (facets) through particular connectors (projectors). This mechanism allows construction of systems where very little is hard-coded at compile time, and a lot is left to configuration, which can usually be performed by a domain expert rather than a software engineer.

1. INTRODUCTION

A software architecture should realise, in terms of architectural components, crucial concepts that are of use to application domain experts.

First, the architecture must rely on concrete "generic" mechanisms that allow components to be defined and to be composed into a system. Second, both components and composition rules must be specialised to fit specific application domain models. This paper presents a software architecture for monitoring and control systems.

Monitoring and control systems cover a huge range of application areas, from classical process control, to environmental monitoring, urban traffic control, monitoring of historical buildings, and many more. They also share common features, from the point of view of both the domain model and of architectural requirements.

In terms of the application domain model there are two major issues:

1. Physical objects of the environment are represented at several *abstraction layers,* spanning from the field interface layer (sensors and actuators) to higher layers where abstract images of the physical objects live and control-related activities (computation, decision, and presentation) are performed;
2. There are two major *information flows*. The *observation flow* starts from the actual state of physical objects and updates more and more abstract images. The *control flow* starts from the expected state of abstract images and updates more and more concrete images – ultimately, it controls the state of the physical objects.

This scheme is quite general, and accommodates particular cases. For instance, in a pure monitoring system the observation flow only exists, whereas in an open-loop control system the control flow only exists.

In terms of concrete architecture, there are two major requirements:

1. modularity and configurability: the domain engineer must be capable of building, configuring and managing his or her system by exploiting reusable components that model domain-level concepts, without having to deal with implementation-related concerns
2. behaviour: the domain engineer must also be capable of defining the dynamic behaviour of the system – in particular, of specifying the timing of the observation and control flows – without knowing the internal structure of the components nor the idiosyncrasies of the technological platforms.

The Kaleidoscope architecture we present in this paper is a general framework (Gamma, 1995) that attempts to meet the above requirements of monitoring and control systems. The key issues, described in detail in the rest of the paper, are the following:

– Application domain entities are modeled by *conceptual images*.
– A conceptual image is realised by a set of *concrete images,* possibly hosted by different physical nodes.

- The alignment of information between concrete images is performed by *projectors*.
- The overall behaviour is defined by *strategic*[1] *components* that drive the projectors.

The major advantage of the Kaleidoscope architecture is that not only does it provide a sound basis for building a distributed system by composing domain-related components, but it also raises up to the programming-in-the-large level the definition of the strategies that drive the alignment of the concrete images and the execution of computational activities. In fact, individual components do not embed specific strategies, which can be easily defined by the domain engineer according to specific domain requirements.

A detailed description of the architecture is given in parts 2 to 6, and some implementation issues are set forth in part 8. The Unified Modeling Language (Fowler, 1997) (Penker, 1997) is employed to describe modeling and design issues, while the Java Programming Language (Gosling, 1997) has been chosen as the reference implementation language.

2. THE ARCHITECTURAL APPROACH

2.1 Static Architecture

According to (Shaw, 1997), the architecture will be described in terms of *computational components* and *connectors*. Adopting an optical metaphor, a computational component is the *image* of an object in the application domain. Images are defined at two levels.

1. *Conceptual images* model the entities that are meaningful from the point of view of the application domain. A conceptual image defines:
   - a set of *facets* i.e., attributes, that model the information associated with the entity (for example, current value, average, variance, etc.);
   - a set of *filters* i.e., methods, which are responsible for converting the information among different facets.
2. *Concrete images* are subclasses of the conceptual images that can be regarded as views of the latter. They provide concrete representations of an abstract image, according to both application requirements and physical deployment issues.

In general, a concrete image implements a subset of the facets and of the filters defined by the conceptual image; moreover, a conceptual image may not be associated with all the possible types of concrete images. Some

[1] To avoid misunderstandings, we shall use the term *control* in the sense of "process control", and *strategy* to denote the policies which drive the dynamics of the system from the software architecture point of view.

standard types of concrete images, that will be treated in detail in the following, are:

- acquisition image
- peripheral processing image
- central processing image
- persistent image
- presentation image
- simulation image
- actuation image

In the remainder of this paper, unless explicitly stated, "image" will denote "concrete image."

The connectors, on their part, play the role of *projectors*, which align the information contents (facets) of the images and encapsulate system-dependent communication issues.

For example, suppose a traffic engineer wants to control the access to a certain area of a town in order to avoid congestion. After installing the physical devices (e.g., photocells and traffic lights), he or she instantiates one acquisition image per photocell (and/or one actuation image per traffic light), plus one peripheral processing image for each device (which will be hosted by a peripheral node). Also instantiated are some images that will reside at the central control room, namely a central processing image, a persistent image that has the goal of permanently storing data, and a presentation image devoted to data visualisation. Moreover, the traffic engineer instantiates the projectors that will be in charge of aligning the proper facets of the images. The resulting static architecture is sketched in *Figure 1*.

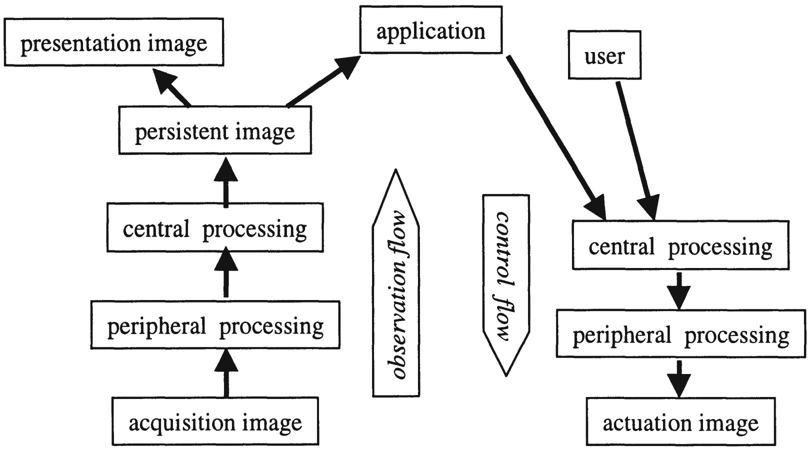


Figure 1. The static architecture

2.2 Dynamic Behaviour

Information (in the example, the presence of a vehicle detected by a photocell) will be first acquired in the form of facets local to the acquisition component, then "projected" onto the different images, and "filtered" to obtain more abstract facets (counts, averages, statistics, graphics, and so on) This is the *observation flow*.

At the end of the observation flow, information will eventually reach the user, be it a traffic operator exploiting the presentation image, or an application component relying on the central processing image. The user can monitor data and exercise control by modifying the central image of the traffic lights (e.g., by changing the desired plan of "red" and "green" phases). The changes of the central images are projected and filtered, and eventually reach the actuation images that physically control the traffic lights. This is the *control flow*.

The traffic engineer is in charge of specifying the dynamics of the system. The heart of the dynamic behaviour lies in the updating of the images (which may imply filtering actions). Consequently, the projectors play a major role in the definition of the dynamic behaviour.

A key point of the whole architecture is that *neither images nor projectors embed any activation strategy*: they can be viewed as passive entities (Tisato, 1996). They define data (facets), operations upon them (filters), and alignment mechanisms (projectors), but they are not aware of all the issues regarding *when* to retrieve data, transfer them between images, and filter them. Such issues depend on specific application domain requirements and must be defined by the domain engineer without having to know the internal structure of the individual components (images and projectors). In other words, images and projectors are aware neither of the static structure nor of the dynamic behaviour of the system; this peculiarity can be summarised by saying that the components have "no implicit architecture" (Cazzola, *et al.*, 1998a).

The control of the overall dynamics is up to specific *strategic components* (also named *strategists;* see part 6), which can be selected and parameterised by the domain expert. This way, strategic components are kept strictly separate from images and projectors, and they can be easily re-used under different strategies in a different context of the same domain, or in a completely new domain. To summarise:

- Conceptual images model concepts that are meaningful in the application domain.
- Concrete images model specific views of conceptual concepts, which are specialised based on the kind of processing they perform and on their deployment.

- Facets are the actual containers of data.
- Filters model manipulation of the data of a concrete image.
- Projectors align different concrete images of the same conceptual image.
- Strategic components drive the activities of projectors and filters.

3. INFORMATION COMPONENTS

The information components lie at the heart of the system's static architecture, as they are the actual managers of application data. There are three types of information components: conceptual images, concrete images, and facets.

3.1 Conceptual Images

Modeling complex domains requires a proper use of abstraction to make the system manageable. Conceptual images realise a high-level abstraction of reality, defining which aspects of the domain must be taken into consideration. A traffic control example is the concept of "gate", which captures a flow of vehicles at different levels of abstraction – vehicle presence, vehicle count, average, and so on.

A conceptual image is an abstract class, defining data and methods that will be inherited and implemented by the subclasses (concrete images; see part 3.2); thus, a conceptual image defines the semantics of some monitored and/or controlled data.

3.2 Concrete Images

Conceptual images are abstract classes and, as such, represent abstract concepts; therefore they need to be further specified to be useful. For example, asking if vehicles pass through a gate at a given time makes little sense. What is sensible to ask is how many vehicles passed in the last minute, or how many passed yesterday between 2 and 3 p.m., or how many pass on the average. These different views of the same conceptual image are dealt with separately in the various subclasses called the *concrete images*, and in other classes, associated with these, called *facets* (see part 3.3).

Concrete images are the actualisation of conceptual images. They turn into practice the semantics of the abstract images. More technically, concrete images are subclasses of conceptual images, inheriting a subset of their attributes (i.e., facets; see part 3.3). Their instances are actual software objects deployed to different locations according to their functionality.

Special cases of images are *information sources* and *information sinks*. Sources retrieve data from the outside world and feed them to the system; they provide a “generate” method that, in a system-dependent way, acquires data from physical devices, or generates them itself, typically by simulation. Sinks feed back to the outside world information that was generated by the system; they provide a “use” method that transfers to the outside world the generated data. In this case, too, the way data are output is system-dependent (e.g., graphical report on a workstation, satellite transmission, Web-based diffusion, commands to a physical device, etc.).

Concrete images, albeit architecturally well characterised, do not constrain the designer’s choices as to which functionality each image is to integrate. As an extreme example, a designer might want to incorporate complicated statistical processing into an acquisition image; this would clearly be against the philosophy of the architecture (and good sense too!), but it is reported here just to give an idea of the flexibility of the system.

A basic set of concrete images has been identified, which, in the authors’ opinion, is enough to cover most concrete cases of monitoring and control systems, even if it is always possible for the designer to define new ones, or eliminate some of the existing ones.

- The *acquisition image* is a source image that collects physical data from the outside world. It can be thought of as a device driver of the physical sensor, and resides on the same node. For instance, it may represent the state of a photocell (vehicle passing/not passing). According to the type of sensor, different types of acquisition image may exist, and possibly some device-dependent code will have to be integrated.
- The *peripheral processing image* resides on a peripheral processing node and supports a first level of information abstraction. For instance, it might include a facet modelling a short-time count of vehicles passing in front of a photocell. In the symmetric case of a traffic light, it might include a facet modelling the desired duration of the next “red” or “green” phase.
- The *central processing image* provides an abstract view that supports user interfaces and major computational activities (statistics, optimisation, control decisions), and typically resides on a central processing node. It includes facets suitable for these tasks.
- The *persistent image* is a permanent view filed in a database. It typically includes facets that model timed sequences of primitive data. Once the time series have been stored, all sorts of computations can be performed on them, including statistics, data mining, etc.
- The *presentation image* is both a sink and a source image. As a sink image, it exports to the outer world the data acquired and processed by the system. We purposely use a vague term such as “exporting” because

it makes no assumption as to which medium will be used to communicate the information to the outside world, be it a graphical workstation, a satellite, or the Web. In this respect the proposed architecture is completely open; one may think of a simple Web server that, via CGI o servlets, delivers static snapshots of the system state, or of a complex, three-tier, full-fledged presentation server that in real time delivers the presentation image. Note that the projector-based approach (see part 6) allows the timings related to data transfer towards the presentation image to be decoupled from the timings related to inner data transfers. On the other hand, the presentation image may also be a source image, in that it exports methods that allow the user to exercise control. For example, the presentation image (i.e., the GUI) for a photocell will present the actual number of vehicles passing through a gate, whereas the presentation image for a traffic light will provide a slider to tune the duration of the "red" and "green" phases.

- The *simulation image* is a source image, situated below the peripheral processing image (i.e., at the same level as the acquisition image) or, possibly, below the central processing image. Its goal is to supply data that are not retrieved by direct acquisition. The rationale behind the existence of such an image is twofold. First there is *fault-tolerance*: in case of unavailability of the acquisition image (due to hardware or software failure), the simulator works "on-line" and can feed the system with reasonable data, thus ensuring the continuity of service. Second, there is *decision support*: it is possible to perform what-if analyses, based on the "off-line" execution of the simulator. As we shall explain later, the switch between acquisition mode and simulation mode can take place at run-time in an immediate and transparent manner, by simply activating the proper connector. This way, the system can work in a non-stopping mode.
- The *actuation image* is a sink image whose goal is to translate user commands and decisions taken by the processing components into real actions performed on the domain. Note that both the acquisition image and the actuation image are in fact nothing but drivers of physical devices: sensors and actuators respectively. This symmetry does not exist by chance; it is the expression of a substantial architectural equivalence between monitoring and control. In both cases, information output by the system (monitored data in the one case, commands in the other) is acquired, processed, and finally output by the system in the form of presentation (monitoring) or actuation (control).

3.3 Facets

Facets are the actual containers of data, and as such are the items of interest to the end user. Typical examples of facets are current value, average, and variance. They are the actual realisation, in concrete images, of the attributes found in conceptual images. In UML terms, an image is an aggregate of facets. The choice not to directly include these data into the concrete images may seem strange, but it stems from the intention to keep the architecture as general and flexible as possible. Incorporating the facets in the form of attributes of the concrete images would mean hard coding the semantics of the images. Using the proposed approach, however, the designer is free to include in each concrete image only the facets he or she is interested in. This approach allows insertion of new classes to meet particular requirements (for instance, advanced statistical processing) that obviously cannot be all provided *a priori*; in this case, the designer must simply define the facet classes he/she is interested in and associate them with the concrete images that will utilise them.

As facets are the actual data, it is these objects that are transferred through the projectors when images are aligned (as explained in part 6). Of course, the same facet may have different implementations for different images that may be hosted by different nodes and rely on different platforms. In this case, filters (see part 5) are in charge of performing marshalling. In order to ensure coherence of observations, every facet has a timestamp.

3.4 Aggregate images

What has been presented so far refers to monitoring and control of single, elementary entities. It is nonetheless clear that a general reference architecture must contemplate the possibility of monitoring and controlling images relative to complex entities that model significant domain concepts, but do not correspond to the direct abstraction of physical devices. For example, a traffic operator would like to reason in terms of "urban areas", that can be observed (in terms of global number of vehicles in them) and controlled (in terms of incoming and outgoing vehicle flows).

In order to model such situations, the concept of *aggregate image* has been introduced, which models, at a higher abstraction level, a set of elementary images. Clearly, it is not possible to associate a physical sensor with a whole zone. Thus, aggregate data cannot be directly derived from information coming from the outside world, but need to be computed from elementary data that are themselves inside the system.

The structure of an aggregate image closely resembles that of elementary images, so there will exist a conceptual image that specialises into concrete images, and a set of facets that are associated with the various concrete images. The main difference between aggregate and elementary images lies in data acquisition (and, symmetrically, actuation). While elementary images acquire data directly from the outside world through the acquisition image (or by simulation), aggregate images acquire data inside the system, namely from the elementary images the aggregate image refers to. This yields two main consequences:

1. An aggregate image is an aggregate of the elementary images it refers to.
2. An aggregate image is not associated with an acquisition image (nor with an actuation image), since acquisition (actuation) is performed by processing data associated with elementary images.

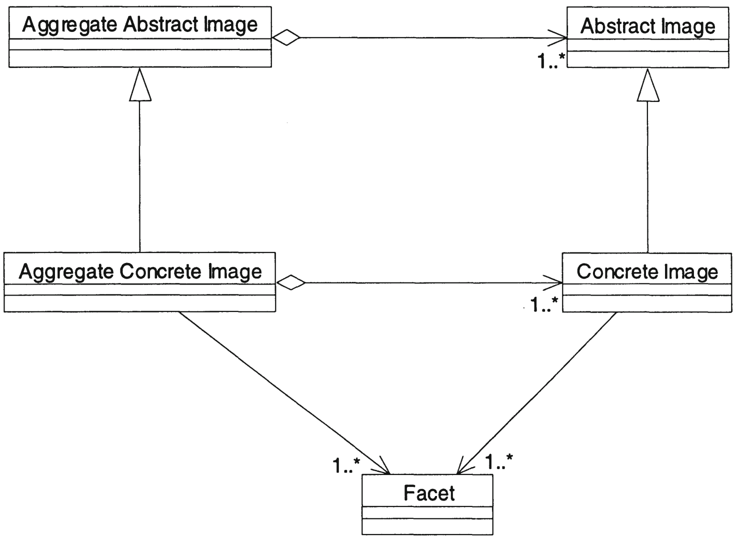


Figure 2. The relationships among classes

Aggregate images can be put at different levels of abstraction. Suppose that, in order to avoid network overloads, the designer chooses to perform aggregation at the peripheral processing level. The following scenario will occur:

- The aggregate image has no associated acquisition image.
- The peripheral processing aggregate image locally collects data from the peripheral images of the elementary entities it aggregates.
- From this point on there is no more actual difference between an aggregate image and an elementary one.

– It may be the case that elementary entities have no higher-level images (central processing, presentation, and so on).

The schema described so far, and illustrated in *Figure 2*, constitutes what may be considered a design pattern (Gamma, 1995).

4. PROJECTORS

Projectors are in charge of linking images, and aligning the information they host. According to this definition, a projector would seem to be nothing more than a communication channel; on the contrary it corresponds to the concept of a connector (Shaw, 1996), which has a much richer meaning and much deeper implications.

A connector is an abstract entity, completely independent from its actual implementation, that can be mapped onto the most diverse hardware and software communication architectures. Moreover, a connector, beyond its communication capability, can also perform information retrieval, i.e., extraction of information from the components it is connected to. Communication being the main task performed by a connector, a set of communication-related attributes and methods can be collected in a class, that will act as a superclass for all classes implementing connectors. This class is abstract, thus it only serves as a base class.

Projectors are in charge of communication among concrete images. Being specialised kinds of connectors, projectors are themselves abstract entities, independent of their actual implementation. At one extreme, there might be a projector implemented as an almost empty object; at the other extreme, a projector might implement a complex application protocol.

Despite their variety of possible implementations, projectors still retain a set of common characteristics that have to do with their main function i.e., alignment among concrete images. To this aim, they are provided with two standard methods (called respectively "alignUp" and "alignDown") whose task is to realise the transfer of facets among concrete images, as we shall further explain in part 6. That transfer will take place upstream (alignUp) to realise monitoring and downstream (alignDown) to realise control.

Note that projectors perform an inter-image alignment of uniform facets, i.e., they transfer the state of the same facet from one image to another. In other words, if two images are connected by a projector, the same "projected" facet must be included in both images. No computation or conversion is ever performed by the projector (such conversions are performed by filters; see part 5). For example, both peripheral and central processing images of a photocell contain a "short-term-vehicle-count" image, which is aligned by a projector.

The symmetry between monitoring and control (see part 2.2) is now fully realised: each of these two activities is implemented exactly the same way, by alignment of concrete images. For monitoring (method "alignUp") data are transferred from sensors to the central parts of the system, for control (method "alignDown") commands from the user are transferred towards the periphery of the system, and finally to the actuators.

5. FILTERS

Filters are particular components associated and co-resident with, concrete images.

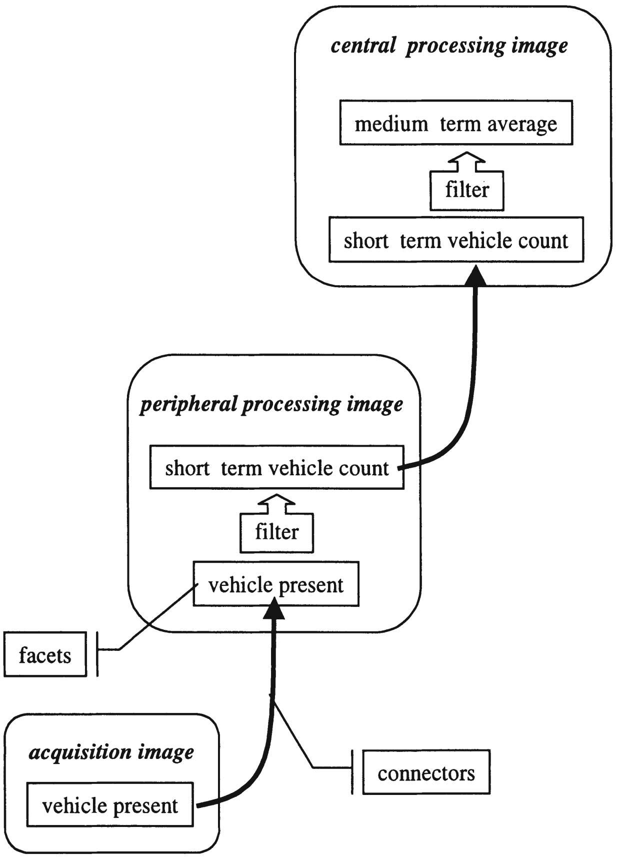


Figure 3. Facets, projectors, and filters

Their task is to transform information between different facets, perhaps performing data conversions if the same facet has different implementations in different images. Just as connectors are responsible for an inter-image alignment of uniform facets, filters are responsible for an intra-image conversion among different facets. For example, as shown in *Figure 3*, the peripheral processing image of a photocell has both a "vehicle-present" and a "short-term-vehicle-count" facet. The former one is aligned according to the corresponding facet of the acquisition image, whereas the latter one is updated by a local filter according to the changes of "vehicle-present". Accordingly, the central processing image has both a "short-term-vehicle-count" and a "medium-term-average" facet.

In general, filters should be triggered by strategic components (see part 6). In many cases, filters are triggered as a consequence of information transfer performed by projectors, in which case an automatic triggering mechanism can be exploited.

6. STRATEGIC COMPONENTS

6.1 Goals

As pointed out in part 2.2, *strategic components* manage the execution of system activities. They perform the following tasks:

- Decide when to activate acquisition images (through the "generate" method discussed in part 3.2) to acquire data from the physical sensor.
- Decide when to align concrete images activating the appropriate projectors (through the "alignUp" and "alignDown" methods).
- Decide when to invoke filters. In fact, in case of very frequent updates, it may not be efficient to re-compute every time the statistical quantities, as this activity can be complex and time-consuming.
- Decide when to export information to the rest of the world (through the "use" method of the sinks), thus obtaining the visualisation or printing of data, in the case of the presentation image, or the activation of control devices, in the case of actuation images.
- Decide when to re-compute new plans (see part 6.2).
- React to asynchronous events generated by the concrete images (see part 6.3).

6.2 Architecture

The architecture of the strategic subsystem is somewhat orthogonal to the rest of the architecture. In the following we sketch the basic features of the strategic subsystem of Kaleidoscope. It is organised into a set of *strategic components* linked by *strategic connectors*, according to a strongly hierarchical model in which each layer corresponds to a different abstraction layer, and higher-level strategic components control lower-level ones.

Strategic components are instances of a Strategist class (see (Cazzola, *et al.*, 1998b)) that can be specialised into subclasses. Instances of this class are linked by special kinds of connectors, called *strategic connectors*, which are also subclasses of the generic connector class.

All communication between a strategist and the non-strategic components it manages takes place locally; in other words, the two components reside on the same processing node. This reduces network traffic and allows hard-real-time problems to be managed locally.

Each strategist, in order to perform its task, needs an action plan, i.e., a list of the form <timeInterval, className:methodName>, that specifies the time interval in which a certain method of a given class is to be invoked. These plans can be passed on as parameters to the strategists; *this allows a change of plans at least at configuration-time, or even at run-time*, rather than at compile-time, thus rendering the system much more flexible.

Each strategist has the ability to execute various plans: this way, when faced with events or particular situations, the component can change plan at run-time without the need to stop and restart the system. To that aim, each component is equipped with a "changePlan" method that takes as a parameter the new plan identifier. Of course, a component cannot change plan before the current one has come to a "safe" situation. This yields the necessity of providing plans with breakpoints, i.e., points at which it is safe to interrupt plan execution. Given the hierarchical structure of strategists, the "changePlan" method is invoked by the upper-level strategist.

6.3 Strategies

Since Kaleidoscope wants to be a *general* architecture, it should accommodate both a time-driven and an event-driven model. The authors definitely do not want to be involved in the decade-long event-driven vs. time-driven debate (Tisato, 1995), so the basic architecture makes no assumption as to which triggering philosophy will be chosen. This is possible because *all the non-strategic components, be they images or projectors, are passive*, thus no strategy-related elements are embedded within them. The designer can freely decide to build the strategic subsystem on either a time-driven or an event-driven approach (an example will be provided in part 7).

The specific architecture we proposed in part 6.2 is basically time-driven, and relies on the execution of timed plans. In many cases, plans can be defined at configuration time. For instance, photocells are sampled at fixed time intervals, the alignments and the decision activities are triggered at fixed intervals too, and the states of the traffic lights are controlled via predefined plans.

Asynchronous events can be taken into account either via a "plan-selection"-based policy or via a "plan-formation"-based one. In the former case the strategist reacts to an event by choosing a plan among the predefined ones. In the latter case it defines a new plan. Once again, the designer is completely free to choose whatever policy he or she prefers. As a guideline, the authors suggest that a default "emergency" plan be activated upon receipt of an alarm. Such a plan has the goal of governing the system in a consistent and safe, if not optimal, way. While the default plan is executing, it is always possible to re-compute, as a non-critical background process, a new optimal plan.

7. DYNAMIC BEHAVIOUR: SCENARIOS

Let us now look at a possible execution scenario, referred to as the traffic control example. The scenario is sketched in *Figure 4*, which highlights (as thin dotted arrows) the triggers of the actions, not the underlying strategy. For simplicity, the figure shows some generic "strategists" and does not consider the allocation of components to nodes, even if the strategist shown corresponds to a reasonable allocation.

The execution scenario depicted in the figure proceeds as follows:

a) A strategist decides that it's time to acquire data and calls the method "generate" of the photocells acquisition image, which gets information from the physical sensor and inserts them into the appropriate facet ("vehicle-present").
b) A strategist decides to align the "vehicle-present" facet of the photocells peripheral processing image, by calling the "alignUp" method of the appropriate projector. Note that this activity might have a timing that is different from the previous one – and might be controlled by a different strategist. This may be necessary to face the case of heterogeneous hardware and software devices, which are very likely to have very different updating intervals and modes.
c) A strategist triggers the intra-image filtering that updates the "short-term-vehicle-count" facet of the peripheral processing image, as explained in part 5.
d) A strategist decides to align the "short-term-vehicle-count" facet of the central processing image.

e) A strategist triggers the filter that updates the "medium-term-average" facet of the central processing image.

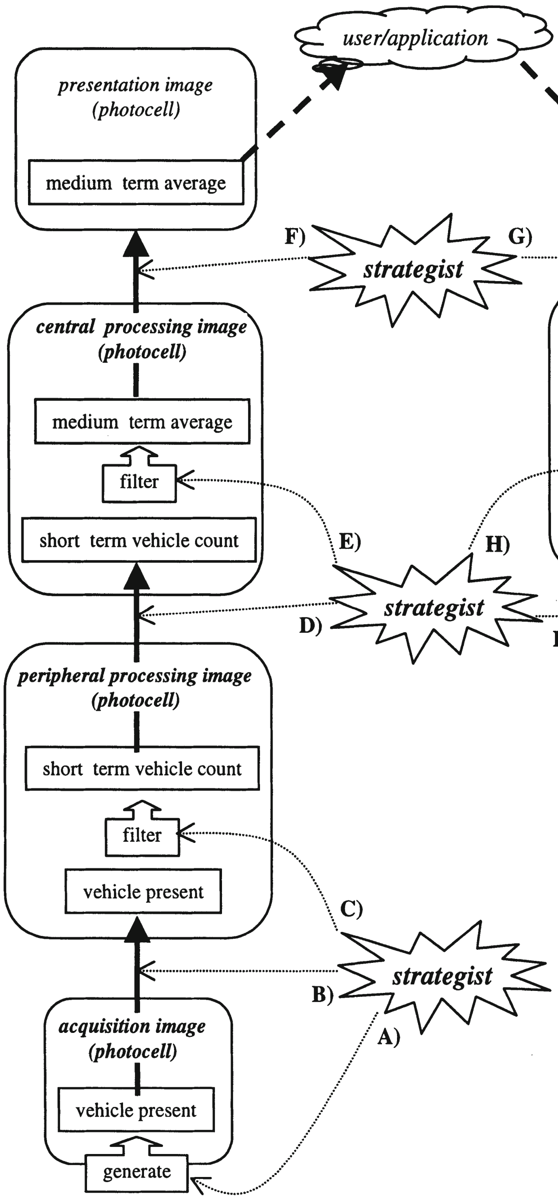


Figure 4. Execution scenario

f) A strategist decides to align the "medium-term-average" facet of the presentation image, which is presented to the user (be it a human being or a process control application).
g) Eventually the user changes the "desired plan" facet of the traffic light presentation image, and a strategist decides to align the "desired plan" facet of the traffic light central processing image.
h) A strategist triggers the filter that updates the "red/green duration" of the central processing image.
i) After that, the control flow proceeds till the physical traffic light is changed.

As pointed out before, different triggers may have different timings, according to specific application requirements. Timings are defined by strategists, and neither images nor projectors are aware of them. Several scenarios can be devised.

– fully time-driven: all the actions of both connectors and filters are triggered by the strategists on the basis of timed action plans (see part 6.2). This is the scenario for a traffic control system oriented to medium-term optimisation, where sampling the system state and tasking decisions at a fixed rate is a major requirement.
– "upward" event-driven: the actions are triggered by the strategists as a consequence of events that asynchronously notify a state change of some acquisition image. This is the scenario for a system that must manage alarm conditions (incidents and the like).
– "downward" event-driven: the actions are triggered as a consequence of events generated by a presentation image – i.e., by a user. This is the scenario for a system that allows an operator to freely observe and control the traffic system.

Of course, mixed strategies can be conceived. We note that all the strategies can be implemented by changing the strategic components only, without any change to the re-usable components, i.e., images and projectors.

Simulation can play an important role in making the system reliable and in ensuring continuity of service. Should a physical sensor break, the strategic component can activate a simulation image and trigger a projector that gets information from the simulation image rather than from the acquisition one.

8. IMPLEMENTATION ISSUES

In part 2.1 we said that a concrete image inherits methods from its base class "conceptual image," while attributes are rendered as separate objects, namely facets. Note that the concept of "inheritance" we are referring to in this context is a very abstract one, meaning that, at a very high abstraction level, concrete images share the semantics introduced by a conceptual image. Obviously, this does not mean that concrete images must be implemented as actual subclasses of conceptual images; in fact, we already stated that, in order to implement attributes of the conceptual image, facets are used, which are not attributes in the usual sense.

Another issue relative to facets is that when the strategist commands an alignment, the projectors retrieve the appropriate facets from within images. In general, this implies marshalling, therefore a projector must invoke a filter. In order to make such an approach possible, each concrete image must maintain a data structure (presumably a hash table or something similar) containing the symbolic names and the addresses of the facets it includes. Moreover, each filter is equipped with a method, called "read", which is called by the projector, retrieves from the concrete image the reference to the proper facet, instantiates a copy of the facet ,and passes it on to the projector. A "write" method of the filter performs symmetric actions.

This procedure requires rather sophisticated mechanisms to avoid unpleasant situations, such as having to hard-code in the application code the mechanisms to recognise the kind of facet the projector is carrying. Such hard-coding would cause the resulting code to be non-extensible and not adaptable to new situations, which would be against the aims of the architecture. Vice-versa, one can think of using mechanisms such as computational reflection (Maes, 1987) or run-time type investigation in order to retrieve at run-time the type of the facet, and to be able to install a new one of the same type. (Such a mechanism is already implemented in the Java programming language (Gosling, 1997).)

Filters are an interesting example of the role abstraction plays in a software architecture such as this. In fact, at general design level, they are defined as stand-alone classes; however, proceeding further down to detailed design, one realises that a filter may actually be rendered as a distributed object that lies partly within the concrete image it is bound to, partly in the facets, and partly in the connector it is referred to.

As we said in part 2, images, strategic components, and projectors are strongly separate entities This means that images and projectors never know who is controlling them, while the opposite always holds, i.e., each strategist knows the identity of all the images and projectors it is controlling, and thus knows exactly what kind of notifications can be sent to it. This way, in order to implement default plans, it is sufficient to provide an association table of the form <notificationCode:planToExecute>. Upon receipt of a notification, the strategist will adopt the associated plan.

To solve the opposite problem (how can the image know to whom it must issue the notification?), one can think of an "implicit invocation" architectural style (Shaw, 1996), also known as the “Model-View-Controller” design pattern (Gamma, 1995). The strategist declares itself interested in a certain set of notifications, and these will be delivered to it. (Such mechanism is already implemented in Java (Gosling, 1997), under the name of “observer/observable”.)

9. CONCLUSIONS AND FUTURE WORK

This paper presents a software architecture that can be used as a basis for the design of a wide range of monitoring and control systems. The architecture relies on a set of reusable components (*images*), connected by particular connectors (*projectors*), while all strategy-related activities take place in separate components *(strategists).* The proposed approach brings consistent benefits to both software engineers, who can build re-usable components according to the reference architecture, and domain experts, who can build a system without having to be software experts.

The current activity regarding the Kaleidoscope architecture is twofold.

1. A prototype is being implemented in Java to test implementation alternatives and check the soundness of the architecture.
2. A specialisation of the architecture to specific domains is in progress. In particular, the urban traffic domain is being treated; the system is essentially modelled as a set of intersections and arcs, represented as conceptual and then concrete images and by the relative facets. Other conceptual images model meaningful entities such as traffic lights, sensors, and in particular accumulation points, i.e., abstract containers of vehicles that can be recursively defined. Specific presentation images support a Web-based information export to citizens and patrols (police, fire brigade, etc.) deployed on the territory.

The preliminary tests confirm the soundness of the approach.

REFERENCES

Cazzola, W.; Savigni A.; Sosio, A. and Tisato, F. (1998a), A Fresh Look at Programming-in-the-Large, *Proceedings of The Twenty Second Annual International Computer Software and Application Conference (COMPSAC ’98)*, August 1998, Austria, Vienna.

Cazzola, W.; Savigni, A.; Sosio, A. and Tisato, F. (1998b), Architectural Reflection: Bridging the Gap Between a Running System and its Architectural Specification, *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering and 6th Reengineering Forum*, March 8-11 1998, Florence, Italy.

Eriksson, Hans-E. and Penker, M. (1997), *UML Toolkit*, John Wiley & Sons, 1997.

Fowler, M. and Scott, K. (1997), *UML Distilled: Applying the Standard Object Modeling Technique*, Addison Wesley Object Technology Series, 1997.

Gamma, E.; Helm, R.; Johnson, R and Vlissides, J. (1995), *Design Patterns: elements of reusable object-oriented software*, Addison-Wesley, Reading, MA, 1995.

Arnold, K. and Gosling, J. (1997), *The Java Programming Language*, 2nd edition, Addison-Wesley, 1997.

Maes, P. (1987), Concepts and Experiments in Computational Reflection, *Proceedings OOPSLA '87*, Sigplan Notices, ACM, October 1987.

Shaw, M. and Garlan, D. (1996), *Software Architecture. Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

Tisato, F. and DePaoli, F. (1995), On the Duality between Event-Driven and Time-Driven Models, *Proceedings of the 13th IFAC Symposium on Distributed Computer Systems*, Toulouse, France, September 27-29, 1995.
Revised version also published as:

DePaoli, F. and Tisato, F. (1996), On the complementary nature of event-driven and time-driven models, *Control Engineering Practice - An IFAC Journal*, Elsevier Science, June 1996.

DePaoli, F.; Tisato, F. and Bellettini, C. (1996), HyperReal: a Modular Control Architecture for Real-Time Systems. *Journal of System Architecture*, Vol. 42, N. 6-7. December 1996.

Segregating the Layers of Business Information Systems

An interface-based approach

Johannes Siedersleben[1], Gerhard Albers[2], Peter Fuchs[1], Johannes Weigend[1]
[1]University of Applied Sciences of Rosenheim, Marienberger Str. 26, D-83024 Rosenheim, Germany; +49/8067/9122 (phone), +49/8067/9123 (fax)
[2]Software Design & Management, Thomas-Dehler-Str. 27, D-81737 Munich, Germany; +49/89/63812-0 (phone), +49/89/63812-150 (fax)
johannes.siedersleben@t-online.de

Key words: Business information systems, external representations, interfaces, layers, reusability

Abstract: This paper presents a refined layered architecture for business information systems of any size. It allows a strict separation of application logic, database access, and user interface and is largely independent of programming languages, database management systems, operating systems, and middleware.

1. INTRODUCTION

1.1 Business information systems

Business information systems (e.g., systems for order processing, stock control, or flight reservation) are used daily by many people and are crucial for a company's business. The focus of this paper is front office systems as opposed to back office systems like data warehouses. The systems considered here can be characterised as follows:

- They are individually designed and implemented for big companies (telecommunication, railroad, travelling, car production). It takes more than one calendar year and several dozen man years to implement them. They contain at least several hundred thousand lines of code.

- Their class (data) model contains several thousand attributes and several hundred classes (entities). They handle many gigabytes of data..
- They cope with high transaction rates, the transactions being short and relatively simple.
- They run in a heterogeneous environment: A system might involve several programming languages (e.g., Java, C++ and Cobol), several database management systems (e.g., SQL Server and DB2) and several operating systems (e.g., MVS, Unix and Windows NT).
- Their expected lifetime is 10 years or more.

In spite of many valuable results in the area of software engineering, the design of these kinds of systems has proved to be difficult. This paper presents a standard architecture, based on defined interfaces between components, that simplifies the design of these systems. It distils the experience of several dozen software projects in which the authors have been involved.

1.2 Why is software design so hard?

1.2.1 No metric for software design

We all know that software should be easy to maintain, easy to extend, easy to reuse, open to additional features, and fast. These properties are hard to measure (performance excepted), hard to achieve, and some of them are contradictory. A crucial feature like extensibility can at best be defined in terms of examples. There is no beaten path to a defined degree of maintainability, extensibility and so on; it all depends on the intuition of the system architect. The degree of maintainability actually reached by a given project is visible only after many years. It is not measured by quantitative means, but only assessed by the naked eye.

1.2.2 The three layer architecture does not work

The three layer architecture (see Ambler, 1998; Denert, 1991) is a well established recipe for the design of business information systems:

- the dialog layer controls the interaction with the user
- the application kernel implements the business logic
- the database access layer takes care of all database accesses

There has been little change to this scheme during the last few years; variants being discussed in the area of workflow systems do not affect the key ideas of this architecture. The intended benefit is the separation of concerns:

- The application kernel is neither aware of the user interface nor the database; changes to the these are transparent to the kernel.
- Dialog and database access layer have limited knowledge of the application. They are not aware of the business logic.

Experience shows that it is hard to keep the details of user interface and database off the application kernel. Two phenomena are frequently observed:

1. The business logic moves from the application kernel to other layers; the application kernel just vanishes.
2. The application kernel gets polluted with details that should be hidden in other layers.

There is a blatant lack of standards for the interaction of layers. Numerous projects have spent many years designing and redesigning these layers' responsibilities. To our experience, the intended benefit of the three layer approach never materialised to the expected degree.

1.2.3 Too many APIs

The software community is literally flooded with new technical APIs and new versions thereof: JDBC, ODBC, OCI, ADO, OLE-DB, AWT, MFC and so on. This makes the software architect's job even harder: Which API can I rely on? Which one works? How many workarounds will be necessary? How expensive would it be to migrate from – say – ODBC to OCI or vice versa? Furthermore there are some old-fashioned, awkward-to-use but very reliable host APIs (e.g., BMS, IMS, VSAM) that will not disappear in the near future and often have to be taken into account even with new systems. How can we cope with this variety of different APIs of different ages? For small systems with a short lifetime, these questions are of little importance. Our concern, however, is big systems with an expected lifetime of 10 years or more. These systems must carefully encapsulate all technical APIs.

1.2.4 Where to go?

Why are some components reusable and extendable, but others not? There is one obvious observation: Software that deals with many different things at a time is bad in all respects. The programmer's nightmare is return codes from different technical APIs mixed up with application problems, and all that within a couple of lines of code. This idea can be formalised: Any business information system is concerned with the application domain (this is why it is built), and technical APIs like operating systems, database management systems, and middleware (no system can run in thin air). Therefore, the components of a given system can be divided into four disjoint categories of reusability. Any piece of software can be:

1. determined neither by the application nor by technical APIs
2. determined by the application, but not by technical APIs,
3. determined by technical APIs, but not by the application,
4. determined by the application and by technical APIs.

The term " determined by" can be read as "knows about," "depends on," or "is influenced by." Code determined by the application knows about business objects like customers, accounts, flights, or aircraft. Code determined by technical APIs knows at least one API like ODBC or OCI. For the sake of convenience, we mark software determined by the application with an "A" and software determined by technical APIs with a "T", thus yielding the four categories 0 (neutral), A, T and AT.

0-software is ideally reusable, but of no use on its own. Class libraries dealing with strings and containers (e.g., STL) are examples for 0-software. 0-software implements an abstract concept, such as a dictionary or a state model. Note the difference between a class library like STL and a technical API like MFC that acts as an interface to a lower-level API (Win32). Using STL means choosing an abstract concept (namely that of containers, iterators, adapters, etc.) and works wherever C++ runs. Using MFC excludes all environments MFC does not support.

A-software can be reused whenever the given application logic is needed as a whole or in parts. Other applications access A-software typically via middleware like CORBA, DCOM or RMI.

T-software can be reused whenever a new system uses the same technical environment (e.g., JDBC, ODBC, AWT or MFC). One nice feature of T-software is that its size increases sub-linearly with respect to the number of business classes. A cleverly designed and carefully written technical component that works fine for 20 business classes can do as well for 200. In fact, JDBC (and other APIs as well) does not care at all about the number of business classes that are using it.

AT-software is hard to maintain, reluctant to change, can never be reused, and should hence be avoided. The architectural quality of a software system is inversely proportional to the share of AT-code. Unfortunately, at least at a small scale AT-code is easy and straightforward to write. Quality software is characterised by the complete lack of AT-code and by clean interfaces between 0, A and T. This is where we should go. It goes without saying that there are major management issues to the question of reusability that this technical paper does not address.

2. QUALITY SOFTWARE ARCHITECTURE

2.1 Overview

It is possible to define a standard architecture that contains some 0-components, no AT-components at all, and which establishes clean interfaces between 0, A and T components. This architecture is being developed by a project at Rosenheim University of Applied Sciences (Germany) in cooperation with Software Design & Management, a company in Munich. Its name is QUASAR, from "quality software architecture."

QUASAR employs the terms "use case" and "business object" in the sense of Jacobson (1997) with the following refinement: A use case seen as a software module knows which steps have to be performed in which order for the use case to succeed. A step of a use case can be any operation on business objects or on other use cases. A use case can be persistent (stay alive for days or months) or transient. There is no clear distinction between a use case and a business object. A flight reservation may be regarded as a use case, a business object, or both at a time; it is up to the designer to choose. QUASAR makes minimal assumptions about the design of use cases and business objects. QUASAR's concern are reusable components that are called by the application and which call it back.

QUASAR's mission is a standard architecture for business information systems that significantly simplifies the design. That means that there are reusable components of a defined category with defined interfaces, and running prototypes in evidence of feasibility that can be used as-is or as templates for project-specific implementations. The remainder of this paper describes the current state of our work.

2.2 Central themes

The central themes of the quality software architecture are

- QUASAR attempts to be as non-intrusive as possible. An application using QUASAR has to implement defined interfaces and to call others. All assumptions are laid down as interfaces. In particular, there is no superclass from which all business objects or use cases have to be derived.
- QUASAR is open to almost all programming languages. There is a focus on object-oriented languages, but QUASAR components can be implemented in C or Cobol as well. QUASAR doesn't rely on language features like RTTI (C++), Java reflection classes or Smalltalk blocks.
- QUASAR shields the application kernel from technical APIs (like OCI or ODBC) by means of a stable, vendor-independent interface.

- QUASAR components can be used independently.
- QUASAR avoids code generation. Often code generators tend to be slow, unreliable, do not generate what is really needed and turn out to be a burden for the development process.
- QUASAR only performs error and exception detection. The handling is left to the application.

2.3 Architecture

QUASAR's backbone are business objects and use cases. The main menu of most systems can be thought of as a special use case which gains control at system start-up. We call this the use case controller. Normally, a use case is started by its constructor or by a special start method.

On the right hand side of Figure1 there are three components: The **Workspace**, the **DataStore** and the **concrete API,** which provides access to the database (ODBC, OCI,..). The DataStore hides that API behind a generic interface which can be talked to in terms of DataContainers. Thus, it is unaware of use cases and business objects. The DataStore interface provides the usual find/update/insert/delete operations.

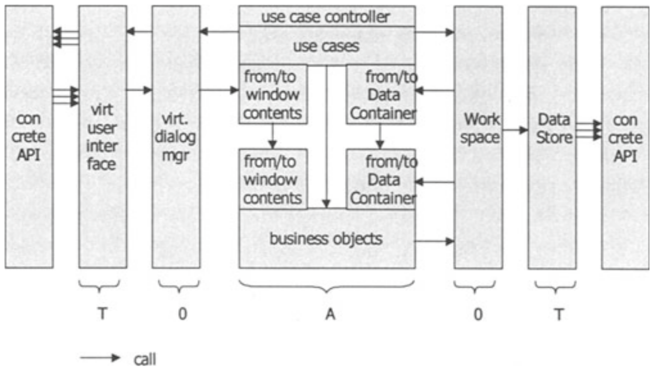


Figure 1: QUASAR architecture

We use the term "persistent object" for all business objects and all use cases that are to be stored in the database. In general, all business objects and some of the use cases will be persistent. A persistent object cannot be stored as such, but only as a DataContainer. So, each persistent class has to provide methods to map the object onto its representation as a Data-Container and vice versa. In most languages, these methods will have names like "toDataContainer" and "fromDataContainer"; in C++ it is simple to overload the shift operators.

Following Java naming conventions, the interface that defines these methods is called "Storable"; all persistent classes must implement it. In section 2.6, we sketch how this can be done. The Workspace links the application to the DataStore. Its interface follows closely that of DataStore, but is defined in terms of Storables. Thus, it is the Workspace that calls the Storables' mapping methods. The Workspace takes care of object identity and implements a given transaction strategy (optimistic or pessimistic, see section 2.6). The important thing to note is that the Workspace is 0-software. Its use could become as obvious as that of – say – a container. The communication between the application and the database uniquely relies on two interfaces: The Storable interface with its to- and from- methods and the Workspace interface. This is the only link between the two worlds; there are no assumptions about each other except those cast in the two interfaces.

In a typical implementation, there will be exactly one instance of Workspace and one instance of DataStore for each human user logged in. Variants of this rule are hinted at in section 2.6.

Let's look at the left hand side of the figure. There is a symmetry not only in the figure but in the whole way of thinking. We will see that accessing a database and accessing a user interface have a lot in common.

Again there are three components: The **virtual dialog manager** (VDM), the **virtual user interface** (VUI), and the **concrete API**, that provides access to the physical screen, which can be anything in the area of BMS, Motif and MFC. The virtual user interface hides that API behind a generic interface that can be talked to in terms of virtual windows and virtual widgets. Like the DataStore, the VUI is unaware of use cases and business objects and knows basically only two classes: virtual windows and virtual widgets.

We use the term "presentable object" for all business objects and all use cases that are to be presented to the user. In general, all use cases and most of the business objects will be presentable. A presentable object cannot be presented as such, but only as a virtual window. In complete analogy to the database side, each presentable class provides the methods "toVirtual-Window" and "fromVirtualWindow". Of course, the corresponding interface is called "Presentable". The virtual user interface presents virtual windows by means of its central method "processVirtualWindow". Within that method, it handles incoming events. Many of them can be dealt with directly by the VUI, for example, field editing. The main benefit of the VUI is its ability to condense physical events (e.g., "button X released", "field Y changed") into virtual events. Virtual events are abstractions of physical events, e.g., "analyse user input", "confirm" or "cancel". In the simplest case, a physical event is directly mapped onto one virtual event. In general, there can be arbitrary definitions like "button Y released and field Z changed and field T not changed". This idea works for graphical user interfaces as well as for block-oriented ones: In a 3270 environment, there are very few physical events ("key *K* is hit, where *K* is one of ENTER, PF1, PF2,...") which can be easily refined to many different virtual events. Whenever the virtual user interface recognises a virtual event it calls back the virtual dialog manager and tells it to process the virtual event.

The virtual dialog manager acts as a link between the use case that wants to execute its dialog and the VUI that communicates with the concrete API. Each use case has a corresponding instance of VDM, which in turn has an instance of a VUI. Thus, for each active use case, there will be one VDM instance and one VUI instance. Each VDM instance manages one dialog only and dies when that dialog is closed. The dialog management is controlled by an interaction diagram. Interaction diagrams have been used for quite a while and have proven to be extremely useful for the precise description of user interactions; see Denert (1991) for more details. Here, the use case hands over an instance of an interaction diagram to its VDM instance. Thus, the VDM constructor expects a presentable object (i.e., the use case) and an interaction diagram as arguments. It calls the use case's toVirtualWindow-method and transmits the result to its VUI. When the VDM instance is called back with the processVirtualEvent method, it consults its interaction diagram and decides what to do. Frequently the ruling use case is called back with its fromVirtualWindow method.

The VDM is little more than an interpreter of interaction diagrams and fairly easy to implement. It is, of course, 0-software. The to/fromVirtual-Window methods are far less straightforward; see section 2.7.

An active database would call back the DataStore very much like BMS/CICS or Motif would call back the VUI. Both the VUI and VDM can be hierarchically organised along the lines of the PAC pattern (Bass, Coutaz, 1991).

2.4 Virtual devices

Virtual devices encapsulate technical APIs. We have seen two of them:

1. the virtual user interface (VUI), hiding APIs like BMS or MFC
2. the DataStore, hiding, for example, ODBC or OCI

Additional virtual devices can be introduced for any technical API that should be hidden from the system (e.g., workflow systems or archives). Virtual devices don't know anything about customers, accounts and orders; instead, they deal with virtual containers containing virtual items. VUI is concerned with virtual windows and virtual widgets; DataStore manages DataContainers and DataContainerColumns. The definition of a virtual container consists of three parts: definition, contents, and context, which can be implemented as a class or a record each, depending on the programming language. In order to bring a virtual device into being, you define the interface, the item, the container, and you implement the interface for at least one concrete API. This is, of course, a T-implementation (determined by exactly one API). It is written in the same language as the API, that is, a DataStore implementation for ODBC is likely to be written in C or C++; a JDBC implementation in Java. A virtual device only knows a handful of classes.

It is up to the designer to determine the amount of information known by virtual devices. Choosing a complex virtual container allows full exploitation of the features of the physical device but makes the from/toVirtualContainer methods expensive to implement and reduces portability. Choosing a dumb virtual container guarantees portability, allows for cheap to/from-methods, but a 3270-minded virtual window won't look very beautiful on a Motif screen. This is not as harsh a problem as it might appear at first sight: Many famous standard products (e.g., SAP) have a graphical user interface that is almost completely form based and could be implemented by means of a rather dumb virtual window.

2.5 DataStore

This section describes the DataStore interface and its implementation with a relational database in mind. However, it is equally possible to have the interface implemented for an object-oriented database or for VSAM. Why would somebody choose not to directly access an object-oriented database but rather via a DataStore? There could be at least two reasons:

1. In spite of the ODMG efforts, the available object-oriented database management systems differ significantly. Many vendors are small companies whose future is hard to predict. Within the context of a reengineering and/or migration project, a given application may switch from a relational database to an object-oriented one, or, even worse, may have to access both of them at a time. So, for many applications it is crucial to separate database-influenced code from application logic.
2. Even with object-oriented databases, database classes and application classes are not necessarily identical. Performance considerations at the database level should be invisible at the application level.

It depends on the information conveyed by the DataContainer if the DataStore-implementation is able to exploit the actual database's features.

Let's look at a relational DataContainer, that can be mapped onto one or more physical rows of one or more tables. The **DataContainerDefinition** is a data structure (class or record) that contains all information necessary to talk to the database about rows of a particular table (field name, data type, length, precision, primary keys, etc.). It is up to the software architect to allow few or many data types in the definition. The DataStore maps these virtual data types onto the physical ones.

The **DataContainerContents** is a container of column contents that in turn are just values of the corresponding data type. It contains a reference to the corresponding DataContainerDefinition that tells the DataStore how to read the contents. It is crucial to make sure that a given contents matches the definition it refers to.

The **DataContainerContext** contains additional information for the DataStore about how to process given contents. When writing to the database, it can be important to know which fields are unchanged; when reading, perhaps not all fields are requested. DataContainerContext is a container of column contexts that convey state information such as "changed/unchanged" or "requested/not requested".

The DataStore interface defines the ordinary database operations in terms of DataContainer definition, contents, and context. The most obvious operations are:

```
DsReturnCode find(DataContainer dc) throws DsException;
void update(DataContainer dc) throws DsException;
void insert(DataContainer dc) throws DsException;
void delete(DataContainer dc) throws DsException;
```

The DataStore also supports the database's transaction logic:

```
DsReturnCode commit() throws DsException;
void rollback() throws DsException;
```

It depends on the chosen transaction strategy if we need an additional operation for locking:

```
DsReturnCode lock(DataContainer dc) throws DsException;
```

The DataStore's return codes (class DsReturnCode) are used for normal events (e.g., find() didn't find anything, commit() encountered a collision with another user); exceptions are raised for unexpected events (e.g., database not available). The DataStore interface provides at least one operation for bulk reading.

```
DsResultSet findMany(DataContainer dc) throws DsException;
```

This is a query by example: findMany accepts an example and searches all matching DataContainers. The DataContainerContext tells the DataStore which fields are to be read from the database. This operation returns an instance of DsResultSet, that implements the usual Collection interface, but can be finely tuned with respect to prefetching and caching. It is accessed by an iterator; the actual database fetch operation may happen at any time between the invocation of findMany (earliest possible point in time) and the dereferencing of the iterator (latest possible point in time). A variety of similar findMany methods can be implemented accepting more than one example connected by logical expressions. Experience shows that almost all queries of a typical OLTP application can be dealt with in this manner. The DataStore also supplies DDL-methods, so it can check at runtime whether the actual database layout matches the actual classes.

The DataStore sketched here is easy to implement, especially when it can be copied from a template. However, it does not provide the full query functionality of SQL or OQL. If this is required there is an obvious work-around: a findMany() method that directly accepts an SQL or OQL query string. However, this pollutes the application kernel, effectively transforming it into AT-software, so the workaround should only be a well documented, rarely used hack.

2.6 Workspaces and storables

The Storable interface is implemented by each persistent class:

```
void toDataContainer(DataContainer dc);
void fromDataContainer(DataContainer dc);
void resolve(Workspace ws, DataContainer dc);
Oid getOid ();
Storable clone();
```

The to/from methods have been discussed already. They are easy to program: The to/from methods of a complex class call the to/from methods of its components; the DataContainer itself knows how to handle elementary data types (`int`, `float`, and so on). This is an application of the well-known streams concept that is used similarly by, for example, XDR (external data representation) or NDR (network data representation). The to/from-methods map the object onto its database representation and vice versa. For example, several object fields may be combined into one database field. This is why in general it is not a good idea to have these methods generated.

The resolve method resolves the object's references: It knows the foreign keys contained in the DataContainer and calls the Workspace's find-method in order to get an object reference or a container of references. The clone-method is needed for technical reasons.

Inheritance is easily dealt with if the following rules are observed:

- There is one DataContainer for each persistent object regardless of the number of superclasses contributing to that object. The DataContainer contains a discriminator indicating the actual class.
- The toDataContainer method of any derived class first calls the toDataContainer method of its superclass (like a constructor).
- The fromDataContainer method of any superclass calls the fromDataContainer method of the actual derived class using a switch-statement on a discriminator contained in the DataContainer.

It is the DataStore's job to map the DataContainer according to the DataContainerDefinition onto one huge table (one table per inheritance tree), many tables (one table per class) or anything in between.

The Workspace interface is almost identical to the DataStore interface except that Workspace deals with Storables whereas DataStore only knows DataContainers. If we implement DataContainers as Storables, then any Workspace implementation automatically implements DataStore and can be used anywhere a DataStore is expected.

The Workspace's primary task is to call the appropriate to/from and resolve methods. It has, however, further reasons for existing:

- It implements object identity, that is, subsequent finds yielding the same object return a reference to that object, not a copy. The Workspace needs the getOid() method in order to identify objects.
- It implements a given transaction strategy. The optimistic strategy reads without lock and checks only at update time if there was a collision with a different user; the pessimistic strategy locks all objects on read.

The idea of the Workspace is "What you say is what you get." All changes of objects sharing the same Workspace are immediately visible to all those objects that represent together an area of integrity. It is only at commit time that these changes are published to the DataStore behind the Workspace. Pursuing this idea a bit further, one can imagine an arbitrary tree of Workspaces, each managing an integrity area and communicating by means of the publisher/subscriber pattern.

Another extension of Workspaces could handle (not implement!) the 2-phase-commit protocol. A Workspace could have several DataStores hiding different databases. The Workspace commit would be translated into the well-known prepare-to-commit/commit loops . This works fine and is easy to implement if all databases involved understand that protocol. The benefit is that the application is not aware of anything like a 2-phase-commit.

A Workspace should be written in the same programming language as the application and should be directly linked to it. A client/server cut between application and Workspace can be compared to accessing the STL by – say – CORBA. But a client/server cut between Workspace and DataStore is a very natural matter. It is perfectly possible to have the application written in Java and to provide fast access to Oracle by a DataStore written in C and using OCI. All you have to do is to transform DataContainers between Java and C. There are many ways of doing that using sockets, JNI, RMI, CORBA, or combinations thereof.

2.7 Virtual user interface (VUI)

The virtual window contains information about things to be displayed. A virtual window can be displayed as (part of) a physical window or distributed among several physical windows. Depending on the concrete API, the VUI provides callback methods for all physical events it understands. It is up to the designer to make virtual windows very intelligent (e.g., know about things like tree views) or rather dumb (e.g., 3270-based). At any rate the virtual window contains:

- The definition of virtual events. It is the VUI's main job to map physical events upon virtual ones thus keeping the other components free of knowledge about screen, mouse, keyboard, and other devices.
- The data types of the input fields. The VUI performs all field related type checks. The more data types a given implementation knows the more checks it can perform. If a given VUI implementation encounters a data type it doesn't know, it calls back its VDM.

The central method of the VUI is "processVirtualWindow". Within this method, the VUI listens to physical events until it recognises a virtual event in which case the VDM is called back. There is one instance of VUI for each active dialog.

The mapping between an object and its representation as a virtual window is similar to the mapping between an object and its representation as a DataContainer. For example, it is up to the VUI to decide in function of the screen size to represent a given virtual window as one or more physical windows (that is, a window provided by the concrete API).

2.8 Virtual dialog manager (VDM)

The virtual dialog manager is instantiated every time a use case decides to execute itself. Its main ability is to present Presentables (objects implementing the to/fromVirtualWindow-methods) and to process virtual events when being called back by its VUI-instance.

It manages the dialog by means of an IAD (interaction diagram). This is a finite state machine that controls the states of a dialog. For each state, there is a set of legal virtual events. The IAD indicates which action is to be executed when a given virtual event occurs and it defines the resulting state in function of the outcome of that action. These actions may be known to the VDM itself (e.g., close_window) or they are methods of the ruling use case. The use case has to register these methods with the IAD. In general, most of the dialogs will be covered by just a handful of standard IADs.

2.9 Virtual windows

A common argument against this kind of architecture says that implementing virtual windows amounts to reimplementing the widget hierarchy of AWT, Motif, or whatever. This can be avoided by using concrete widgets directly within virtual windows, thus accepting use cases that are no longer A but AT. When designing virtual windows, it is important to know whether or not there is a binary link between use cases and VUI. If so, the field's data types, for example, can be simply given as interfaces the VUI calls back each time a field is edited. Likewise, the concrete widget classes can be used directly by the use case. If not, all information in the virtual window has to be coded as strings interpreted by the VUI. This latter choice is obviously suboptimal as far as performance is concerned, but it is ideally suited for a client/server cut between VUI and VDM. For example, it is possible to have a VUI implemented as an applet talking to a remote application (including VDM, use cases, business objects) written in any language into which the string based virtual window can be translated.

3. BENEFITS

Let us summarise the main benefits of QUASAR:

- Virtual devices only know about virtual containers. Hence, it is very convenient to have a client/server cut between a virtual device and the remainder of the application. The IDL contains only a handful of class definitions (the virtual container and its items). As a general rule you shouldn't have many business objects on both sides of a client server cut: Maintaining consistency can be a nightmare, even with CORBA.
- It is not hard (even without CORBA) to translate a virtual container from one programming language to another. This is obvious for Java and C++, for example, but can also be done between C++ and COBOL. It is possible to have a VUI written as a Java applet talking to an application

written in C++ talking to a DB2 database via a DataStore written in COBOL.

- Virtual containers can be dumb or intelligent. A dumb container can easily be mapped onto an intelligent one; the other direction is harder, but often possible (an OK-button can be represented as a yes/no input field). Virtual containers could be standardised: The software community doesn't need more than two or three of each kind.
- Implementation of use cases and business objects is not affected by any technical API. There is a direct transformation from the application class model (given in – say – UML notation) to the implementation classes.
- The database design determines the to- and fromDataContainer methods and nothing else. Any change of the database layout only affects these mapping methods.
- The user interface design determines the to- and fromVirtualWindow methods and nothing else. Any change of the windows layout only affects these mapping methods.

There are two important points beyond the QUASAR story:

1. We, the community of software designers, badly need well-defined interfaces between the layers of the classical architecture or variants thereof. Every working day there are many thousand software designer thinking about basically the same design problems. There **must** be an answer to that!
2. Sooner or later the tremendous, unfiltered amount of new technical components will drive us crazy. There **must** be a way to enjoy new features without being forced to migrate complete systems from Java 1.0 to 1.1 to 1.2 to 1.x or from RDO to ADO to OLE-DB or to whatever is cool next week.

REFERENCES

Ambler, Scott W. (1998), Building Object Applications That Work, Cambridge University Press & SIGS Books

Bass, L., J. Coutaz (1991), Developing Software for the User Interface, SEI Series in Software Engineering

Denert, E. (1991), *Software Engineering*, Springer Verlag.

Heuer, A. (1997), *Objektorientierte Datenbanken*, Addison Wesley.

Jacobson, I. , M. Griss, P. Jonsson (1997), *Software Reuse*, Addison Wesley.

INTEROPERABILITY, INTEGRATION, AND EVOLUTION OF SOFTWARE

Security Issues with the Global Command and Control System (GCCS)

Shawn A. Butler
Computer Science Department, Carnegie Mellon University Pittsburgh, PA 15213
shawn.butler@cs.cmu.edu

Key words: Architecture, security, command and control, common operating environment, COE, GCCS

Abstract: The Global Command and Control System (GCCS) was one of the most ambitious and largest software integration tasks in the history of the Department of Defense. As the Chief Systems Engineer for GCCS, I found architectural differences among command and control systems presented unique integration and interoperability challenges. In this paper I present 3 security-related examples of specific problems I encountered when I attempted to integrate several systems into GCCS. I also discuss the problem of system-level security analysis and introduce a framework that software engineers can use to evaluate security.

1. INTRODUCTION

The Global Command and Control System (GCCS) was one of the most ambitious and largest software integration tasks in the history of the Department of Defense. Applications in all stages of maturity were chosen to be integrated into a seamless system, organized around the Common Operating Environment (COE). The COE was a collection of software components commonly found in all command and control systems. As the Chief Systems Engineer for GCCS, I was responsible for every aspect of integration and development including GCCS security.

Security proved the most difficult of all the system integration tasks for two reasons. First, although security specialists talked about the "security architecture" of GCCS, a security checklist derived from a set of security requirements and policies was the best they could produce. Checklists provide a piecemeal approach to system security and usually lack a system level perspective. GCCS interoperability requirements and the process of integrating legacy applications highlighted the role that architectures and system designs played in GCCS security. Second, users' demands for configuration flexibility presented significant challenges to maintaining a consistent level of security with each system. A team of independent security specialists verified the system's security just before fielding. Each security evaluation drained off scarce resources for several weeks at a time. The security team attempted to find security flaws using whatever means they considered reasonable. System security was re-verified each time the configuration of GCCS changed, which was almost monthly during initial fielding.

The Department of Defense relies on a security process that is not compatible with modern software development processes and designs. What I really needed were concrete architectural and design guidance and methodologies for analyzing system security that did not depend on a security specialist's ability to defeat the system after I build it. My frustrations with these two problems led to my current research and the beginnings of a framework to help solve the second problem.

2. BACKGROUND

For many years the Department of Defense operated the World Wide Military Command and Control System (WWMCCS) as the primary command and control system[1]. WWMCCS was a distributed information system that linked major military command centers throughout the world, such as the European and Pacific theaters and the National Military Command Center in the Pentagon. The system processed TOP SECRET, SECRET, and UNCLASSIFIED information, but the bulk of information was SECRET. Since WWMCCS did not have multi-level security, the system operated as if all the information were TOP SECRET. The security requirements for a TOP SECRET system are greater than for systems processing SECRET information.

Military computer security requirements are found in a number of military directives, regulations, and publications. The most well known set of publications are the "rainbow" series, which consist of more than 20 books, each book a different color. The Orange Book defines the concept of

[1] "Command and control" is a term used to define the activity of monitoring, planning and directing military resources.

a Trusted Computing Base (TCB) and specifies the TCB requirements for increasing levels of security. UNIX systems are evaluated and classified based on the criteria established in the Orange Book. Ordinary UNIX systems usually fall into the C1 or C2 class, which is characterized by discretionary security protection requirements. Operating systems classified at the B or A level meet increasingly stricter security requirements and are usually highly specialized operating systems.

The system consisted of 40 Honeywell mainframe computers that serviced numerous dumb terminals within each major command center and in isolated locations throughout the world. Initially built during the 1970's, WWMCCS had become quickly outdated so a modernization program was initiated during the early 1980's (WWMCCS 1992). Research, development, test, and evaluation for the modernization program was budgeted for $773 million, By 1987 the program was behind schedule and over budget so congress cut the FY 88 budget to $21 million. Technology rapidly passed the WWMCCS system and users became increasingly dissatisfied with WWMCCS capabilities. By the mid-nineties most other command and control systems had far exceeded WWMCCS functionality. However, none of the newly developed command and control systems could meet the WWMCCS user's functional requirements.

3. GCCS

The Global Command and Control System, a highly distributed client/server system, was conceived as the replacement for WWMCCS. The initial version of GCCS was a conglomeration of existing command and control applications and new applications that increased and replaced WWMCCS functionality. GCCS consisted of two parts: the Common Operating Environment (COE) and the Application Layer. In order to keep development and fielding costs to a minimum, GCCS consisted of commercial hardware and software and processed only SECRET information. Not only did this simplify the security requirements, but this also meant that GCCS could be fielded on standard commercial UNIX operating systems instead of more secure, and very expensive B2 operating systems. I was responsible for mitigating the risks associated with security weaknesses in the UNIX operating system.

Although most major system development efforts take 5 to 10 years, the Joint Chiefs of Staff wanted the replacement system within 2-3 years beginning in 1994. The primary motivation for the rapid development cycle was the enormous cost of operating WWMCCS, estimated at $7,000,000/month. The 2-3 year development constraint was thought attainable for several reasons. First, GCCS was to be built using existing applications, therefore, GCCS was simply considered an integration exercise, rather than new development. I believe there is a general misconception that integration efforts take less time than new development. Stakeholders assumed that most of the applications selected to be part of GCCS fulfilled enough of the user's requirements that little or no additional development needed to be done. Applications were selected from various Department of Defense agencies and services based on how well they met user requirements and other factors, the least of which was the ease with which they could be integrated, maintained, scaled, or extended.

3.1 GCCS architecture

The foundation of GCCS is the Common Operating Environment (COE), 18 abstract functional components that, when implemented, form the infrastructure services and a set of standard components for all GCCS applications. All existing or legacy applications had to "migrate" to the GCCS COE. Migration required applications compliance with engineering guidance in 4 areas: integration and run-time, user interface, architecture, and software quality. Software for the COE came from each of the services, and the Defense Mapping Agency. I was charged with integrating the COE components and more than 20 legacy applications, all in various stages of development, into a single command and control system that could be uniquely configured at each operational site. GCCS was really a set of command and control applications, which any site could install components as needed.

COE components fall into 3 categories (figure 1):

1. the kernel
2. infrastructure services
3. common support application components

Kernel components consist of the operating system, window libraries (X11R5 and Motif), printing service, executive manager, name service, and a security/system management service. Kernel components are considered essential system components, i.e. every workstation requires these services regardless of function. The security service provides tools to allow system administrators to set up various types of access control accounts. The kernel configuration is tightly controlled since slight deviations from the established configuration could cause disastrous system integration problems. All application developers are expected to develop to the kernel configuration and each developer receives a copy of the kernel and a set of tools to ensure that they follow the run-time integration engineering guidelines.

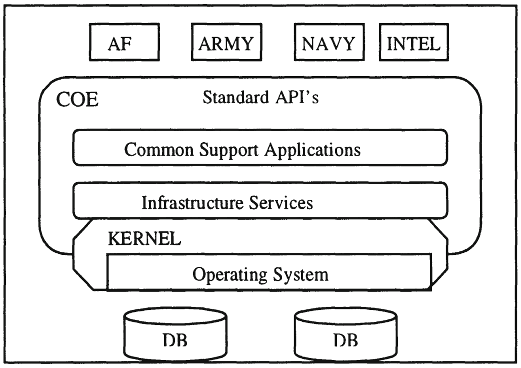


Figure 1. Common Operating Environment

The infrastructure services provide the middleware for the applications. The middleware consists of the following components: management services, communication services, distributed computing services, presentation and web services and data and object management services. Management services are network and system management tools that system administrators use to monitor the system. Distributed Computing Environment (DCE) provided the distributed computing service and the Common Object Request Broker Architecture (CORBA) served as the data and object management service, although the initial GCCS fielded system did not use either service. The communication service provides the interface to external systems. Most external interfaces consisted of messages sent to and from GCCS. Netscape and Internet Rely Chat implemented the web services, but the presentation service was not specified at the time.

The Common Support Layer of the COE consisted of the group of applications that are common to all command and control systems such as office automation applications, situation displays, message generation and management software, etc. At the time, office automation applications, such as word processors, spreadsheets, and slide presentation software did not compare to the products used on personnel computers. UNIX based office automation software had considerably less functionality than PC products. The biggest drawback to the UNIX software was the incompatibility of file formats. Users had hundreds of Microsoft PowerPoint files that were not exportable to the UNIX office automation software and any files created on the UNIX system were not exportable to the PC system. Although PC emulators could have provided a temporary fix to the office automation problem they were too expensive.

3.2 Interoperability issues

The primary drawback of the existing command and control systems was their lack of interoperability between services. Since joint military operations nearly always consist of units from the Marine Corps, Navy, Air Force and Army working together, joint military operations require a command and control system that is interoperable among the other service's command and control systems. As an illustration, many of the frustrations experienced during Desert Storm occurred because systems were not interoperable. Information was frequently exchanged using floppy disks or paper printouts which then had to be re-keyed into an electronic form. As a consequence of the experiences in Desert Storm, interoperability became the number one command and control system requirement in the Department of Defense. Although interoperability was a critical requirement in joint operations, it was not well defined. Interoperability meant different things to different users and under different circumstances. Ideally, systems should be able to efficiently exchange data without any loss of meaning or content, but in practice this is very difficult. The Department of Defense outlines 4 levels of interoperability for command and control systems. The highest level "is characterized by the ability to globally share integrated information in a distributed information space."(DISA 1996). Level 4 was the ultimate goal for GCCS, but each application implemented lower levels of interoperability.

In some cases, application portability across different hardware platforms or operating systems was sufficient to meet interoperability requirements. Data is exchanged because operators from different services are co-located. Each service purchased their own computer hardware so applications built to run on Sun Microsystems hardware did not have to be converted to the Hewlett Packard hardware and vice versa. The lack of portability forced users of one service to learn the other service's application, or for the application and hardware system to integrate with the larger system.

Interoperability could also be achieved if systems could interface using formatted messages, e-mail, or import/export functions. In practice this method was flawed. Currently, all command and control systems communicate with other systems using standard message sets. Unfortunately, the "standard" message sets are not truly standard and not particularly efficient for transmitting all types of information. Many of the message standards were developed before multimedia applications became integrated into command and control systems. Each command and control system selected from several standard message sets, which meant that each command and control system used a different set. In addition, many of the sets were extended with unique messages that were not compatible with the

DoD standard. Interoperability through messages design limits interoperability for two reasons:

1. Users are limited by message content
2. The information is only available when it is sent.

A common view of the battlefield is essential to effective military operations. A higher level of interoperability is required when users shared information from the same source. A common view is ensured when all users have access to the same information source. In practice, different database schemas and data elements made it nearly impossible to share information from a central location. In my experience, integrating databases is one of the most difficult engineering tasks, however, it also provides the greatest interoperability.

3.3 Architectural security issues and interoperability

3.3.1 Interoperability incompatibilities

However interoperability was achieved between two systems, there were usually security implications. If messages were exchanged then encryption of the messages as they pass between two systems was usually sufficient to control access to the information. Encryption incurs maintenance costs because the DoD relies on special hardware for all encryption. The DoD builds many types of encryption devices, all of which are incompatible with each other. No matter which encryption device is chosen, the hardware is scarce and not compatible with other systems that use different encryption devices. Incompatible encryption components make interoperability nearly impossible. This detail is often overlooked when designing command and control systems.

However, when two systems share a common database, then access controls to the database become a primary security concern and incompatibilities between systems can surface. For instance, two applications required access to classified data in the database. One application used database access control mechanisms to ensure that unauthorized personnel did not get unlimited access to the data. Users were restricted from viewing or writing to particular rows, or restricted from certain tables in the database. The other application restricted a user's access to the data by controlling access to the application. Implicit in the latter design is an assumption that any user with access to the application has unlimited access to the database. These two fundamentally different, but valid, points of access control made integration of these applications into a seamless system difficult.

3.3.2 Additional integration problems

GCCS interoperability requirements, integration of legacy applications and the user's demand for configuration flexibility presented significant challenges to maintaining a consistent level of security with each system. Some other security integration problems with the GCS architecture were access control designs and application programming (API) interface mismatches. Access control designs of two systems created a particularly difficult problem. Access controls were usually based on an operator's role or position and the role could change during the operator's shift or an operator may have several roles during the same shift. Problems arose when one system required an operator to log out and then log in when he switched roles, in effect restricting operators from assuming two roles simultaneously. Although this simplified audit trails in that system, it was an unacceptable specification in another system. Security administrators needed the flexibility to accommodate both requirements. Eventually a scheme for access control was developed that was acceptable to all users.

A third problem arose when we discovered incompatibilities between security technologies. Specifically, the Fortezza system developed by the National Security Agency (NSA) was incompatible with Kerberos. Fortezza, NSA's smart card technology, was the latest security mechanism that promised improved system security. NSA considered Kerberos inadequate for GCCS and insisted that GCCS implement the Fortezza system. Although Kerberos had recognized flaws it was available and used in commercial systems. Fortezza didn't have Kerberos' flaws but wasn't available in production quantities.

Furthermore, NSA had not yet developed a Fortezza card that had been adequately tested for SECRET systems. The initial GCCS design used Kerberos and later integrated Fortezza when it became available. Unfortunately, incompatibilities between the application programming interfaces (API's) surfaced, and made integration of the two technologies impossible until the API conflicts were resolved. NSA quickly began to work with members of the Open Systems Foundation, however, the process was expected to take at least two years.

3.4 Architectural integration summary

As the GCCS chief engineer, it was obvious to me that the security of a system does not depend solely on a collection of "silver bullet" technologies and checklists. I could not integrate two systems and plan to overlay the security later. The system security must be designed hand in hand with the system architecture. Interoperability requirements and legacy system integration concerns are not confined to the Department of Defense. As commercial organizations expand and grow so do their interoperability requirements. Companies such as SAP specialize in integrating reusable components. Common system engineering questions include the following:

- What are the design principles and engineering guidance that system engineers should follow?
- How does the architecture support system security?
- What security mechanisms are appropriate for a particular architectural style?
- What are the security weaknesses associated with an architectural style?
- What security conflicts should system engineers look for? What are the design pitfalls?
- How do interoperability requirements affect security?

The list could go on but answering any of these questions would be extremely useful to system developers.

4. SECURITY IMPLEMENTATION

In addition to the architectural issues of integration and interoperability, I was overwhelmed with the myriad security technologies and designs available at the time. While some security solutions were dictated by regulations, I retained a great deal of flexibility to select the mechanisms that constituted the system's security. Frequently, the tension between performance and maintainability and security, raises such questions as: Since GCCS is unusable when full auditing is turned on, how much auditing is enough? What are the alternatives? How does a particular technology fit with other technologies? Are there overlaps, gaps or conflicts? Is the technology right for the GCCS architecture? The most important question for me is "How does a technology affect the overall security of the system? Without this knowledge I find it difficult to make engineering tradeoffs when deciding the right mix of security technologies for the system. System level methodologies or frameworks to analyze security appear to be nonexistent.

4.1 State of the art

Current security models don't seem to support the idea of the system level perspective of security. One of the first security models, the trusted computing base model from the government's Trusted Computer System Evaluation Criteria (Orange Book), was criticized for not addressing network issues and relying on the hardware and software within each workstation to enforce security policies. This model clearly lacks a system perspective. Network models have an implicit boundary that separates insiders from outsiders. Network models emphasize protective barriers that restrict outsiders from penetrating the system, however, there are many internal threats as well. Also, it may be difficult to determine the boundaries of the system in a network model. The "How To" books and trade magazines of security often offer advice along the following lines:

- Identify the system resources that need to be protected.
- Identify the threats to the resources and/or system vulnerabilities.
- Establish security policies.
- Implement cost-effective strategies to minimize the risk threats impose against the resources.

Approaches may vary slightly, but they generally include these four steps. Although the books outline the approach, but they don't really provide any practical strategies. This last step is the kicker. As chief engineer, I found it relatively easy to identify system resources and threats for the GCCS. Implementing cost-effective strategies was difficult because I didn't have a way of comparing alternatives and it was difficult to understand how each alternative fit in the system context.

There has been extensive cryptanalysis research, attempts to discover stronger cryptographic algorithms, and theoretical research in intrusion detection. This type of research is invaluable if we are to rely on these technologies in our systems, but its place in the overall context must be understood. For example, encryption export controls present unique problems when the system must be compatible with foreign military systems. Trade magazines and security handbooks provide high level guidance on how to approach security, and some handbooks such as *Internet Security: Professional Reference* by New Riders Publishing provide very detailed information on how to build a firewall or how to set security sensitive system controls. Threat information taxonomies are easily found in most security textbooks and journals. The Computer Emergency Response Team (CERT) at the Software Engineering Institute (SEI) periodically provides alerts and warnings about security problems and the Internet has a wealth of information about security. How does the system engineer pull the information together to see how all the policies, technologies and design maintain confidentiality, availability and integrity in a system?

5. A FRAMEWORK FOR SECURITY

As Chief Systems Engineer of GCCS, my integration tasks required that I see how each technology, design, or policy fit into the system. I wanted the framework to reveal the system security weaknesses and allow me to see how alternatives compared in the system. I felt such a framework would allow me to make cost-effective decisions about how to choose among all the things I could do to maintain a particular level of security within GCCS. I needed to be able to describe the level of the system security. Such a framework was not available to me at the time. I am now a Ph.D. student at Carnegie Mellon University and have the opportunity to work on constructing such a framework.

Instead of closing this experience with a wish list of questions for researchers to consider, I will lay out a preliminary sketch of the security framework that forms the basis of my own research. The framework takes advantage of the work accomplished by the Networked Systems Survivability Program and presented in *Survivable Network Systems: An Emerging Discipline* (Ellison, Fisher, Linger, Longstaff, and Mead 1997). The following outlines the components of a security analysis approach.

The System Security Analysis Framework (SSAF) is divided into five components:

1. the *system*
2. *security technologies, policies, and design techniques*
3. *known weakness and flaws* for each item described in the security technologies component
4. *threats and vulnerabilities*
5. the *security model.*

SSAF provides a way to include both automated and non-automated security procedures as part of the analysis. The framework accommodates highly connected information systems and standalone systems. It is not constrained by the network topology, nor does it ignore the topology. The security model described in the framework places the system resources at the center of the model and provides a mechanism for showing how the system security mitigates the risk to those resources. The security model pulls all the other pieces together.

1) The *system* component of the framework describes the system architecture, relevant designs, and non-functional attributes. A complete system description that includes how people interact with the system is necessary so that the system engineer can understand how technologies, policies and designs are implemented or fit within the planned implementation. Many of the security technologies adversely impact the other non-functional attributes such as performance, so it is important to understand how the other non-functional attributes will be balanced in the systems. Non-functional requirements such as latency, reliability, and performance, must be identified here. The *system* component provides the context in which the security analysis takes place. Most of the information

for the *system* component can be obtained from architectural description documents, design and requirement documents. Unfortunately, none of these documents were available for GCCS, however, most of the information could have been gathered from developers and software engineers.

2) The *technologies* component is a collection of security technologies, policies and designs that make up the system security. Security technologies include firewalls, access control lists, auditing mechanisms, intrusion detection systems, cryptography, etc. Security policies describe how system privileges are established, processes for reporting violations, password procedures, and any other policy that contributes to the overall security of the system. Configuration settings in products such as access control mechanisms or firewalls enforce many security policies; others are strictly procedural. Each element requires a detailed description about how it is implemented in the *system* described in system section.

3) The *weakness and flaws* component identifies known weaknesses and flaws of each of the items listed in the *technology* component. Security policies often depend on the integrity of key individuals and systems suffer catastrophic failures when an individual betrays his trust. Separate analysis of weaknesses and flaws serves two purposes. First, analysis explicitly raises the awareness of the weaknesses and flaws associated with each item so that the system engineer can address these vulnerabilities, if possible. Second, it identifies areas that might need special attention when the system configuration changes.

4) The *threats and vulnerabilities* component addresses the system threats and vulnerabilities. Almost all security approaches advocate a threat identification step. None of the many threat assessment documents I have read provided specific guidance about threats and vulnerabilities. Documents usually identify a standard set of threats such as vulnerability to electronic eavesdropping, mal content employees, nuclear EMP, and hackers. Reports usually stated that hostile and non-hostile foreign countries might be highly interested in the information the system processed. Some reports might even identify a few flaws in the UNIX operating system for which there were known patches. These reports had relatively little value other than to confirm that I had followed the appropriate procedures and conducted a threat assessment. The *threat and vulnerabilities* component must be much more extensive if it is to be useful.

An initial start at improving threat assessments is a comprehensive taxonomy of threats. Fred Cohen (Cohen, 97) identifies 94 methods of attack. Additional detailed attack information is available from the Internet or from CERT bulletins. Security journal articles offer occasional guidance such as the recent article in Computer & Security (Hancock, 98), which identified several attacks in detail. It may be impossible to collect all of the system threats because there are so many information sources and new attacks are appearing before the old attacks have countermeasures. Developing the *threat* component of the framework will probably be an ongoing process.

5) The core component of the framework is the *security model* (figure 2), which has four layers. The purpose of each of the other components is to help populate each of the four layers of the security model. System threats and vulnerabilities are external to the four layers. Each layer is populated with items from the technologies and policies component. The model is constructed using four defensive layers:

1. protection
2. detection
3. mitigation
4. recovery

Each layer plays a different role in protecting the system resources. Consistent with other security models, the first step is to identify the system resources that must be protected. The *system* component should be the source of resource information.

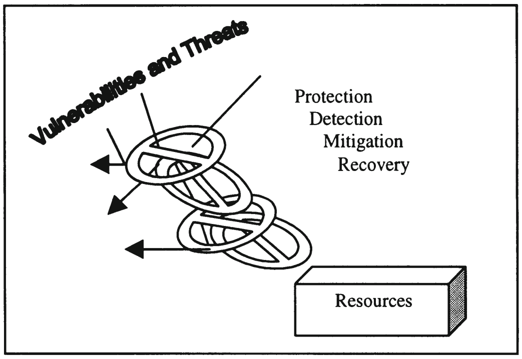


Figure 2. Security framework

The first layer is the *protection* layer. For each threat identified, the security engineer should identify the security technology or policy that stops the threat from gaining or denying access to a resource. This layer should be populated with all the security policies, products and designs that prevent an attack from succeeding. These policies and products have may have flaws, but they may still be effective against some (e.g. accidental) intrusions. Items that most likely fall into this layer are firewalls, passwords, background checks on employees, access control lists, etc. GCCS implemented all of these and more. Ideally, a system engineer would like a one for one mapping between threats and prevention mechanisms.

The second layer is the *detection* layer. Most likely, none of the mechanisms in the protection layer are 100% attack proof. There may not even be a protection mechanism for a particular threat. Hancock (Hancock, 98) identified several attacks, some of which did not have known countermeasures. Without countermeasures, the system engineer needs to identify mechanisms that may detect an attack so that appropriate procedures are developed to properly react to an intrusion or denial of service attack. Intrusion detection systems, virus detection programs, audit trails and logs, special alerts and triggers are all security mechanisms that the security engineer should identify for the detection layer. System personnel should be guided by policy when responding to an attack. For each relevant threat, the system engineer should consider ways to detect an attack.

The third layer is the *mitigation* layer. Here the system engineer considers technologies and mechanisms that minimize the damage an attack may do if it is not detected or contained. System partitioning and system redundancy might be two techniques a system engineer could design into the system to minimize the damage from an attack. The purpose of this layer is to consider techniques and policies that help minimize the damage done from an intrusion that might go unnoticed for some time. Some of the attacks may not cause much damage because they are not particularly destructive attacks, so the system engineer may decide that a particular attack is more a nuisance that doesn't warrant any attention.

Recovery is the fourth layer. The system engineer must be able to recover from an attack. An attack may penetrate the preceding layers so the system engineer should consider how the system can recover from the damage. Back up and recovery procedures fall into this layer. Highly distributed systems like GCCS allow system engineers to design fail over and redundancy into the system without much trouble.

I have only begun to explore the feasibility and potential of this framework. Even if it does not immediately provide the quantitative analysis that most engineers hope for, I think it has potential to compare alternatives relative to one another. It pulls together the essential pieces of information in a uniform, structured way and gives the system engineer a system level perspective.

If this framework had been available to GCCS it would have served us well. The information was available to populate the framework. The GCCS security checklist would have been an excellent starting place to gather an initial list to populate the *security technologies* component of the framework. Also, GCCS security specialists developed a GCCS security policy document that outlined many of the security policies that would be included in this part of the framework. Although these documents were available, there were many discrepancies between the policies identified in the document and those actually implemented. Obviously, it is important to distinguish between the written from the practiced.

6. CONCLUSION

GCCS presented many challenges. Security was the one area in which I felt the most helpless. It seems so much effort is put into each technology and so little effort into the engineering and design principles that need to guide system developers. Trade magazines don't provide the depth of advice that is needed to build the system security from the parts. The research community has not yet produced a model that is of direct, system-level assistance. If we don't understand how security integrates into system architectures today then how will know the role security plays in the domain architectures of the future?

REFERENCES

Cohen F. (1997), Information System Attacks: A Preliminary Classification Scheme, Computer & Security. 16, p. 29-46

Ellison, R.J., Fisher, D., Linger, R.C., Lipson, H.F., Longstaff, T., Mead, N.R. (1997), *Survivable Network Systems: An Emerging Discipline*, Technical Report, CMU/SEI-97-TR-013, 1997

Hancock, B. (1998), Security Views. Computer & Security 17, p. 99-109

Defense Information Systems Agency (DISA) (1996), Defense Information Infrastructure (DII) Common Operating Environment (COE) Integration and Runtime Specification (I&RTS), Version 3.0 (Draft) December 1996

Russel D. and Gangemi, G.T. Sr. (1992*), Computer Security Basics*, Sebastopol, CA: O'Reilly & Associates, July

Modernization of the Worldwide Military Command and Control System (WMCCS), (1992), National Academy Press

Architecture for Software Construction by Unrelated Developers

W.M. Gentleman
National Research Council, Institute for Information Technology, Ottawa, Ontario, Canada.
phone (613) 993-9010, fax(613) 952-0074,
e-mail Morven.Gentleman@IIT.NRC.CA

Keywords: Software architecture, COTS, unrelated developers

Abstract: Suppose one COTS (Commercial Off the Shelf) software supplier provides an interpreter for a problem oriented language, another provides an application generator for producing numerical solvers for a class of partial differential equations, and a third produces a visualization package. A team of domain specialists writes scripts in the problem-oriented language to define cases to be solved, uses the application generator to produce an appropriate solver, solves the generated PDE, and uses the visualization package to analyze the results and adjust the description of cases. Such examples illustrate that large and long-lived software systems can result from the combined efforts of various unrelated development organizations, organizations not even known to one another. No single design authority, to which the others report, has overall system responsibility. Such examples also illustrate the importance of including in software architecture the relationships between entities that exist and are used during the construction process, instead of focusing only on relationships between entities that exist at run time. The needs for software architecture for such systems are not well met by the existing literature.

1. INTRODUCTION

The literature on software architecture, for instance as surveyed in Shaw (Shaw, 1996), has largely focused on components in the sense of computational entities that exist at run time, and their connections in terms of data and control transfer. Various styles of how the same computational system could be structured have been studied, considering how the structure could be analyzed, how individual components could be reused, and how the structure could be reused in other situations. Alternatively, practitioners (e.g., Whitney, 1995; Tzerpos, 1996; Finnigan, 1997) have sometimes used software architecture as a focus on definition and use relationships of entities that exist at runtime. These are, of course, interesting issues, but in many situations they are not the dominant reasons for the architectural structure adopted for the software system. Our definition of the term software architecture is that it is a high-level description of a set of entities and their relationships, the understanding of which is essential to the understanding of the overall structure of the system. This is consistent with the definition other authors have used, although the entities we might consider, and especially the relationships we might consider, are broader than some other authors might take.

In some systems, physical considerations dominate the software architecture. In these systems, any single computer might run several software components, but software entities running on different computers are definitely considered distinct components. The software architecture thus reflects hardware architecture issues such as geographic locality, bandwidth, unique hardware resources, redundancy for reliability, replication for capacity, etc. It may also reflect organizational and administrative realities of the operators, such as what functionality is centralized, what functionality is replicated at each branch plant or even at each workstation, and what functionality is provided by computers belonging to the customers of the system operator, not those of the operator itself.

In this paper we consider situations where the software is implemented not by a single organization, but by a number of organizations, perhaps as a prime contractor with subcontractors, perhaps as collaborating peers with different competencies, or perhaps as suppliers and users of COTS (Commercial-Off-the-Shelf) software products (Dean, 1997; Vigder, 1997). The software development organizations contributing parts of the system may not even be known to one another. These development organizations may contribute parts of the system at very different levels of abstraction.

The situation where the development organizations are unrelated, interacting only as suppliers and customers of COTS software products, is particularly interesting because it is so far from the traditional development model. No single design authority, to which the others report, has overall responsibility for the whole system, in that, by definition, each COTS supplier implements his product to his own specification and timetable determined by his perception as to the market demand, of which this application is typically an inconsequential portion. Detailed specifications and source code for COTS products are rarely available, never mind possible to influence. More seriously, the maintenance and evolution of each COTS component is done to its supplier's agenda, and since obsolete versions usually become defunct, a long-lived system must adapt to the change. When a part evolves and must be reintegrated, the enhancement may not even be implemented by the supplier of the original part — indeed, sometimes a plug-compatible part of completely different design is substituted. Evolution of such systems typically results from the evolution of the different parts, although the introduction of new parts and changes in the relationship of parts can occur. The integrator who brings together all the parts must find a software architecture that can use the COTS products as they are, or as they might be in the future. The integration role may be substantial, or it may be quite small, and may even be automated.

For systems of the kind considered here, there is often not simply a single run time. Often components are run to produce entities, even source code, that will be used by other entities at a later run time. It is thus important in the software architecture also to include relationships between entities that exist, or are used, during the construction process of other entities. Some of the contributions, for instance macro packages, may no longer be localized at run-time, although they may have been localized at some earlier stage in the build process. Potential attributes of a component generated during a run are often not determinable from specific instances generated during particular runs, but may be inferred from the generating subsystem and the input it might be given. Tools used in the build process may be essential in establishing that constraints required at the run time of the application itself are in fact satisfied. The build process itself thus must be part of the software architecture. The system architect plans how the parts are created and brought together. Fortuitously, box-and-arrow diagrams are traditionally used both for explaining the build process and for explaining the software architecture.

These issues will be illustrated by three thinly disguised examples of real systems, systems implemented in the past that continue to be used and to be evolved today.

2. A SIMPLE EXAMPLE

As a concrete example, consider a situation where one COTS software supplier provides an interpreter for a special-purpose, problem-oriented language, another provides an application generator for producing numerical solvers for a certain class of partial differential equations, and a third produces a visualization package. The application of the system might be, for instance, to analyze accidental fires. A team of domain specialists writes scripts in the problem-oriented language to define cases to be solved in terms of geometry, fuels, atmospheric conditions, etc.; uses the application generator to produce an appropriate solver given the characteristics of a specific case; compiles the generated solver; solves the generated PDE for that case; uses the visualization package to analyze the results and adjust the description of cases; and then repeats the cycle. Because the solution of each individual case is a significant investment, and because investigation of an accident involves running many cases and similar cases may show up in future, successful results from each case would typically be stored in an object database, keyed by parameters that characterize the case in the potential search space.

At a sufficiently superficial level, the software architecture is simple and uninteresting: a cycle of subsystems, each producing data for its successor (figure 1).

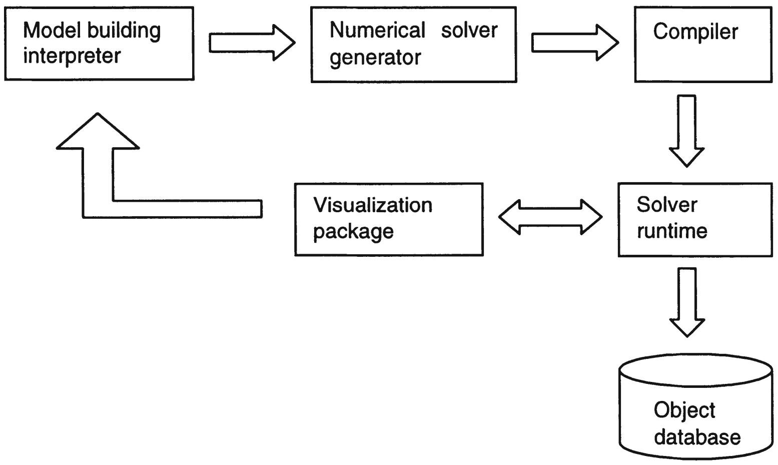


Figure 1. Superficial block-and-arrow diagram for example 1

A deeper level of software architecture elaborates on what connecting to the successor subsystem really entails, on how to exploit previous cases to reduce computational effort, and on how to recover from computational failures such as might result from going beyond the domain of applicability of the physical models or the numerical procedures. Because COTS components produce their output in whatever representation and sequence that they do, and because this is unlikely to conform to the rigid representation and sequence required by the successor COTS component, insertion of, at least, a filter between them is normally required.

In this example, as is often the case, more is needed. The COTS components will not work for every input with which they might be presented, and consequently the architecture must be extended to make provision for exceptions that might be raised. Moreover, a COTS component produces whatever output it produces, and some of this is not actually used by the immediate succeeding component in the notional cycle, but should be passed on through to subsequent components, in the same way that passes in the traditional compiler pipeline burned through intermediate language constructs not operated on until a later compiler pass. Unfortunately, COTS components are unlikely to make provision for simply passing through input that they do not intend to process, so the glue components must facilitate such data bypassing the COTS component. Thus the glue components are normally more general than simple filters.

In this example, the connectors between the components in the superficial view of the architecture are wildly different. At some level this is sufficient, because it shows where dominant relationships exist — and do not exist. At a deeper level we need to understand what they are. The output of the first component, the model builder, is of three different kinds: mathematical formulae which are the partial differential equations and also the description in space of the region of integration; mathematical facts which have been proved or are to be assumed about these formulae; and large numerical arrays that represent initial values, boundary conditions, and other parameter values. Only the mathematical facts and the mathematical formulae, together with a few of the parameter values, are required by the second component, the application generator. This application generator uses these facts and formulae to select among various choices of algorithms and data structures to produce source code for a numerical solver optimized to the particular kind of problem to be solved and the kind of computational resources available to solve it. The first connector thus filters a data stream, possibly reordering typed items and changing their representation. The second connector is a very simple pipe, taking the source code produced by the application system and feeding it to a compilation system. The third connector is more complicated, for it must run the executable image produced by the compilation system, and make available to it the numerical arrays and parameters produced by the model builder. Classically, the fourth connector could be very simple, for numerical solvers wrote their results to files which were later subject to analysis by techniques such as visualization. The visualization might also have required the full output from the model builder. Today, however, visualization is often used interactively to steer the computation as it proceeds, so in addition to inspection of stored partial or complete results, this connector must support debugger-like actions. The final connector, from the visualization package back to the model builder, is simply revising the scripts that define the problems, and is probably accomplished by a standard editor.

3. USES FOR A SYSTEM ARCHITECTURAL DESCRIPTION

By looking at how we might use the architectural description of a system, we can learn more about what it might contain, and how it would be usefully represented. Who is the high level description for?

3.1 High-level description during planning stages

From the literature, one might conclude that the principal use of an architectural description of a system was as a high-level planning document, to agree upon what must be done and what it would be nice to do, then to derive specifications for the components to be implemented. Such a top-down approach can be effective where all the components have to be designed, or even when some of them pre-exist and either the others must be designed to accommodate them or glue must be specified. It can be used to establish properties such as completeness and correctness, and to analyze for properties such as capacity and concurrency. It can be studied for examining dependencies of partial results, and hence for identifying opportunities for phasing computation and so reducing instantaneous demand for memory and other resources. It can be used to study communication requirements between components and hence to assess suitability for distribution in the sense of what should run on which node of a network. If the software system was to be operated jointly by a collection of organizations, the software architecture might be used to study distribution, in the sense of suggesting which components and which responsibilities be given to which organizational units. If the system is to be sold as a commercial product to many different customers, the software architecture might suggest packaging for optional configurations. The software architecture can also serve as a documentation framework, identifying where to record assumptions and dependencies between components.

For software to be implemented jointly by a collection of organizations, a software architecture can provide a framework for considering a number of acquisition and implementation questions which are nontechnical but with potentially technical consequences. What constraints are implied by available components that could be used? Where would separate suppliers of components possibly be effective in reducing cost or improving time to completion? Where does intimate dependency on the same technology imply that the same subcontractor should be used to avoid duplication of startup effort or to avoid errors due to conflicting interpretations? How should implementation responsibilities be divided to correspond to the competencies of different collaborators? And for systems where corporate or national security is an issue, what are the security clearance implications for the implementers of different components?

3.2 High-level description during operation

During operation of the system, the primary use of a system architectural description is tutorial. Because integration of components is often not seamless, the operators of the system often need to be aware of the roles of different components in the production system, and the software architecture often is a useful framework for teaching them. For example, systems often are designed with metering for monitoring and tuning purposes. The significance of such measurements depends on the system architecture, and hence the operator needs to understand the system architecture in order to properly interpret the measurements and act on them. As another example, operational problems often arise in the production use of systems not because of bugs in the implementation, but because intrinsic limitations in the underlying science restrict the domain of applicability, or because choices made during implementation in the absence of knowledge turn out not to be consistent with operational experience. When such problems arise, the operator needs to understand the architecture well enough to recognize the situation and the source of the problem, to take corrective action, and to plan workarounds. Also, as mentioned earlier, the software architecture can be useful for establishing operational responsibilities for different organizational units. Note that operators like this rarely have programming skills.

3.3 High level description during maintenance

Day-to-day maintenance is normally finding and fixing minor bugs, mis-configurations, and interoperability conflicts. Minor enhancements may also be included. For systems that operate nonstop for extended periods, simply monitoring for outages and interpreting logs is often difficult, and the maintainers not only need to understand the software architecture generally but may need to make detailed reference to it in order to localize and eventually isolate errors. Often attempting the repair immediately is not possible, so through knowledge of the software architecture a workaround must be found. Organizing for and actually conducting the repair requires detailed just-in-time learning of the code at the site of the error, as well as at other affected sites. An understanding of the software architecture of the system is key to knowing what to study and the context in which it must be understood. Unfortunately, the skill level of staff employed for this kind of maintenance is often less than that of the initial developers or developers involved in major enhancements.

3.4 High-level description during major evolution

Major evolution of an existing system has much in common with initial implementation, except that because it is incremental there is more incentive to maximize reuse of components from the previous release, as well as to ensure interoperability with data, including control data, produced by or for the previous release. Working out a strategy for actually carrying out the upgrade or replacement of a component is particularly important, especially in nonstop systems. Planning as to how to add a new component or to make other architectural changes is important, and requires a solid understanding of the existing software architecture. That understanding can lead to identification of required competencies and appropriate allocation of responsibilities to carry out the change. However, such changes are relatively rare. The dominant kind of change, especially for a successful architecture, is change by upgrade of a single component.

4. A SECOND EXAMPLE

Another example where the software architecture is dominated by pre-existing components, although not in this case COTS software, is a training system for operators of an embedded system, such as a weapons fire control system, a SCADA (sensor control and data acquisition) system, or a command and control system for air traffic control. For such systems, it is often essential that new operators be trained on the real system, warts and all. Only that way will the new operators get an appropriate sense of the real system's capabilities and limitations, and get the feel of its responsiveness in real time. Consequently, the core component of such a training system is an instance of the real embedded system (see figure 2).

There are three other subsystems in the training system. One is a debriefing subsystem. This is a subsystem that is able to record the student's actions, in real time, as the system responds to interesting situations, so that an instructor can go back through the situations with the student to point out where the student has done well, where the student has used bad judgement or made errors, and what the consequences of these have been. Because real time is an essential aspect of such situations, it is necessary not just to rely on probes into the real system to log the displays produced by the system together with the student's responses to them. It is also necessary to log video and audio of the student's off-line activity, especially where there are several operators working together simultaneously with the system. Many parts of this subsystem pre-exist.

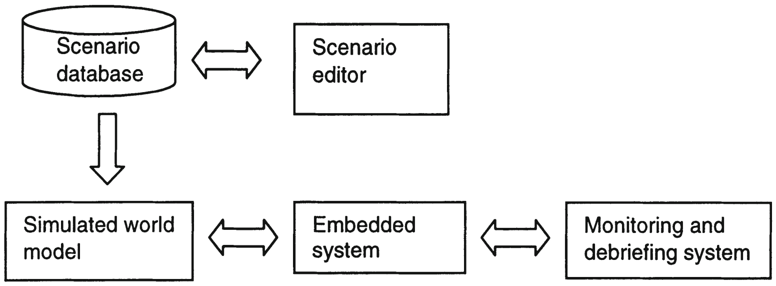


Figure 2. Superficial block-and-arrow diagram for example 2

Another important subsystem is the world modeller. The real embedded system interacts with the real world through various sensors and actuators, and since use of the real sensors and actuators may be impractical for training purposes, they must be carefully simulated. The real sensors and actuators are not independent of each other, but are coupled at least through the real world, so the simulated world for the training system must properly model such interactions. Adequate simulation of the real world requires sufficiently precise modelling of the physical situation, with adequate computational power and typically with a great deal of empirically determined data. It also requires an understanding of what approximations and shortcuts can be taken to meet real-time performance without losing simulation fidelity. Such a simulated world may be a valuable asset that must also be used with trainers for other embedded systems.

Of course to carry out the pedagogical purpose of the training system, the world simulator has to be directed to produce scenarios illustrating situations that the students are to be taught to deal with. Thus the last subsystem is a scenario editor for the simulated world. Obviously scenario development happens at a different runtime than the students lesson. A final wrinkle in such training systems is that qualified instructors are usually in short supply, so the whole training system is partially replicated to allow several students to be trained simultaneously.

This example is interesting because qualifications for implementing each of the four subsystems are quite different. The real embedded system was implemented, and is frequently upgraded, by whoever, typically a systems contractor expert in the sensors and actuators and signal processing. The debriefing system is best implemented by a company well versed in pedagogical techniques, so that it will be easy to capture and to replay appropriate aspects of the student's actions. The simulated world subsystem is best done by a company with strong scientific computing credentials in the appropriate science. The scenario editing subsystem is best implemented by a company that combines usability skills with a clear understanding of what scenarios will be needed. The example is also interesting because at a superficial level, understanding the relationships between the four subsystems is simple. Any attempt to provide a complete and correct description of all the interactions becomes mired in detail.

5. A THIRD EXAMPLE

In next example, the COTS software merely provide a platform on which the system is built rather than performing substantial parts of the computation itself, but limitations of the COTS components are the major cause of architectural choices, with anticipated implementation churn during evolution of these components also playing a role. The system itself is a small but long-lived interactive exhibit for displaying to the public current information about air quality (see figure 3).

Two different kinds of information are presented in the exhibit. The first is descriptive material which is generally static but changes occasionally, for instance when administrative or legal actions affect what is being described. The second kind of information displayed is trends in recent measured data from a network of online monitoring stations. The core of the exhibit is a program written in the proprietary language of a commercial authoring system. This provides facilities from user dialogs to visual effects, and allows the exhibit designer to focus on effective communication with users instead of on implementation.

Unfortunately, the authoring system has functionality deficiencies. The first is that it cannot generate and display the multicolour time series graphs required to display trends. This is solved by a plug-in available from a third-party supplier, together with some glue to remap data structures. The second deficiency is more serious: the network of monitoring stations must be polled by dial-up modem and the measurements accumulated to be shared by several instances of the exhibit, but the language of the authoring system, even with plug-ins, is too weak to support the error handling or concurrency control to do this. The solution is that the exhibit uses read-only optimistic concurrency control to read from a shared database (conceptually a circular buffer of records) maintained by another program. The program maintaining the database is written in another proprietary language, this one being the communications control language of a terminal emulator. The problem of being able to keep the descriptive material up to date without manually updating the whole exhibit each time some fact changes is addressed by keeping all the relevant descriptive material in a database, and using scripts in the database language to walk the database and generate the pages for the authoring system whenever a change is needed. Since the descriptive material includes multimedia items such as pictures, sound and video, an appropriate commercial product is used.

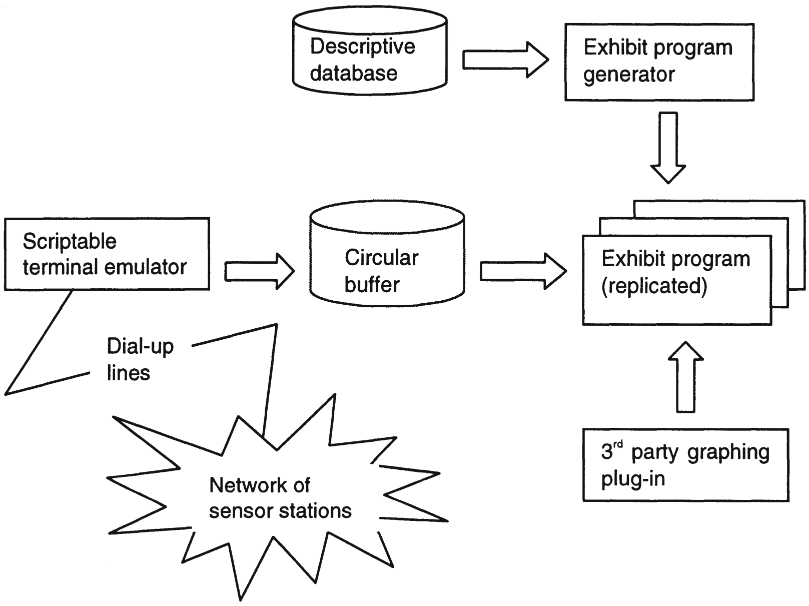

Descriptive database
Exhibit program generator
Scriptable terminal emulator
Circular buffer
Exhibit program (replicated)
Dial-up lines
Network of sensor stations
3rd party graphing plug-in

Figure 3. Superficial block-and-arrow diagram for example 3

The principal use for the system architectural description here is to explain to the front-line non-technical maintenance staff what actions to take when needed. Regeneration of the pages of descriptive material works well as long as maintenance personnel understand they need to update the database, and do not attempt to change the pages directly. Reorganizing the pages calls for different skills, but happens rarely. The monitoring stations have been a continuing source of operational problems: changed passwords block access, station identification is arbitrarily changed, modems go offline for periods stretching into months, stations are shut down and new ones are opened, data format is changed, manual editing of data at the monitoring stations produces records out of sequence, etc. Since the monitoring stations are operated by a different government agency, changes occur without notification and they are not responsive to requests for explanation, much less remediation. Accommodating such situations frequently requires manual intervention, but bullet-proofing the system, so that it reports on detected problems and continues to operate, is mandatory and has architectural implications. The most troublesome problems however have been upgrades to the platforms: the hardware on which the exhibit runs, the operating system on that hardware, and the versions of the various COTS components. These are typically upgraded without notice, and not infrequently the newest versions no longer interoperate successfully. The conflicts are usually easy to resolve, but require technical support. Since technical support is hundreds of miles away and on a time and materials basis, front line support must have a sufficient understanding of the software architecture to localize the problem, perform simple corrective procedures such as reinstalling components, and report symptoms.

6. CONCLUSIONS

Systems with characteristics similar to the examples cited are being developed all the time. The prime purposes of the architecture descriptions of such systems have been for communication with, and analysis by, other people — automated analysis has not been a priority. Architectural styles are not a central issue. For communicating with people, excessive formalism is not necessarily more effective, and text-only descriptions have also proved to have shortcomings. While not entirely satisfactory, the use of block-and-arrow diagrams, supplemented by text, has proved sufficient for the uses cited. What shortcomings have been apparent relate to having consistent presentations of the software architecture at various depths and from different points of view. Too much detail irrelevant to one's current interest is obfuscating.

Perhaps the flaw lies in thinking of the system architectural description as a single document, manually composed, and viewed in its entirety. Instead, we could think of a set of reports generated from a common database (Finnigan, 1997), in the way some re-engineering tools present facts gleaned from existing source code. The central focus would be the cognitive psychology focus of how to make the presentation comprehensible, rather than the computer science focus of how to make the basis general and precise.

In practice, the decomposition into CSCI (Computer Software Configuration Items) for projects constructed under 2167a, and indeed the description of the individual CSCI themselves, often reflected more the competencies of, and relationships between, the prime and the various subcontractors than it did functionality, data access, or allocation of software to hardware. Perhaps this was not so wrong!

REFERENCES

Dean, J.C. and Vigder, M.R. (1997) System Implementation Using Off-the-shelf Software, *Proceedings of the 9th Annual Software Technology Conference*. Department of Defense, Salt Lake City, Utah, 27 April - 2 May 1997.

Finnigan, P. J., Holt, R. C., Kalas, I., Kerr, S., Kontogiannis, K., Müller, H. A., Myloupoulos, J., Perelgut, S. G., Stanley, M., and Wong, K. (1997) The Software Bookshelf, *IBM Systems Journal*, Vol. 30, No. 4, 564-593.

Shaw, M. and Garlan, D. (1996) Software Architecture. Prentice Hall, Upper Saddle River, NJ.

Tzerpos, V. and Holt, R.C. (1996) A Hybrid Process for Recovering Software Architecture, *Proceedings of CASCON '96*, Toronto, ON, 12-14 November 1996, 1-6.

Vigder, M.R. and Dean, J.C. (1997) An Architectural Approach to Building Systems fromCOTS Components, *Proceedings of the 22nd Annual Software Engineering Workshop*. National Aeronautics and Space Administration - Goddard Space Flight Center, Greenbelt, Maryland, 3-4 December 1997

Whitney, M., Kontogiannis, K., Johnson, H. J., Bernstein, M., Corrie, B., Merlo, E., McDaniel, J., De Mori, R., Müller, H. A., Myloupoulos, J., Stanley, M., Tilley, S., and Wong, K. (1995) Using an Integrated Toolset for Program Understanding, *Proceedings of CASCON '95*, Toronto, ON, 7-9 November 1995, 262-274.

Integration of Heterogenous Software Architectures - An Experience Report

Volker Gruhn, Ursula Wellen
University of Dortmund, Software Technology, D-44227 Dortmund, Germany
{gruhn, wellen}@helsinki.informatik.uni-dortmund.de

Key words: Software architecture, database integration, distributed objects, components, migration, software landscape

Abstract: In this article we describe our experience with a software migration project in a telecommunication company. We started from a set of heterogeneous software systems (described by rather different types of software architectures) and we defined a migration path towards an integrated software architecture. On this path several intermediate versions of the software architecture were implemented. We discuss the purpose of these intermediate versions and the problems encountered in the migration path.

1. INTRODUCTION

The experience described in this article is from a project in the telecommunication industry. In this project we analyzed the available software and recognized that several software systems were developed or purchased in a rather uncoordinated manner. In contrast to many other companies (e.g., from the insurance or finance industry), this situation was not due to software systems that were maintained and extended for decades, but was due to a rather fast set-up of software systems for the support of core business processes. The company – as a rather new competitor of the former monopolist Deutsche Telekom – had to establish these software systems in a short term. The systems were developed or purchased by departments independently from each other. Each of these departments focused on its particular problem which had to be solved as soon as possible. The coordination between the departments were not as tight as it would have been in a very well-established business context.

As a consequence of this approach, software architectures of individual systems were driven by different paradigms (some are client/server architectures, others are not; some of the client/server systems are based on SQL databases as distribution paradigm, others are based on transaction processing monitors and others are based on Web servers) [Uma97]. Even worse, the functionality of software systems overlap. For example, master data about customers were administered in several systems. These data had to be reconciled in order to ensure that customers are represented in a consistent way. A mid-term goal was to avoid data redundancy. Another drawback of the initial software architecture situation was that—after a short time of operation—it turned out that higher levels of software system integration were needed in order to provide homogeneous support for the business processes (e.g., same style of user interfaces, same client platforms).

Another organizational issue to be considered was that all systems had to be available all the time (24 hours a day, 7 days a week). The risks of systems not being available after a new release varied from system to system, but, generally speaking, unavailable systems could endanger service delivery, customer satisfaction and business plans.

Another important issue was that the business processes to be supported are supposed to change over time. One particularly critical aspect was that of distribution. Even though most parts of the business processes were carried out at the company's headquarter, the future software system infrastructure should allow for a flexible distribution and allocation to new sites.

Starting from this situation, we developed a migration plan that starts form the existing software infrastructure and ends with tightly integrated software systems realized by distributed objects that support the core business processes of the company. On this path different levels of integration were (and still are) implemented.

In the experience described we put emphasis on different ways of describing the software architectures we encountered in the project. These descriptions vary from rather high-level descriptions of functionality in terms of basic building blocks, to dataflow descriptions of individual software systems, to rather technical descriptions of telecommunication infrastructure needed.

In section 2 we describe the project starting point in some more detail. Next, in section 3 we explain the migration path identified and the different software architectures on the migration path. In section 4 we sum up what problems occurred on this migration path. Section 5 puts our project-specific experience into the broader context of general work on software architecture.

Finally, section 6 concludes with pointing out which compromises were key success factors for the migration described.

2. SITUATION OF SOFTWARE ARCHITECTURES AT PROJECT START

When the project started, several departments were developing complex software systems to support their business processes. Because all these projects were carried out under high time pressure, the underlying system parts were developed and integrated without much coordination. This meant, that many software systems were developed independently of each other as stand-alone solutions without interfaces for data exchange. Some other system parts like administration systems for customer master data or provisioning systems did already exist and had to be reused and integrated into the new business context.

2.1 Initial software landscape

Before we analyzed individual software systems in detail, we examined the overall software situation. At this level, individual systems were considered as black boxes and we only looked at the relationships between the individual systems. The relationships identified were of different types:

- data exchange (pushed by the data-sending system or pulled by the data-receiving system; in each case characterized by the frequency of data exchange)
- access to persistent data
- call relationships

In the following we call this view of the overall software architecture "software landscape." Figure 1 shows the initial landscape and some of the key business objects exchanged between systems. Some details about key business objects are discussed in section 2.2. The software landscape was taken as starting point for the migration. A vision of the future landscape defines the overall migration goal. Thus, any progress can be illustrated in terms of modifications of the software landscape.

Figure 2 describes a more technical view of the software landscape. It shows which software systems are running on which data base management systems and on which operating systems. This view of a layered architecture was very popular in the project, but it turned out to raise many misunderstandings, simply because there was no consensus on the meaning of layers and the meaning of the relationship with neighbouring layers.

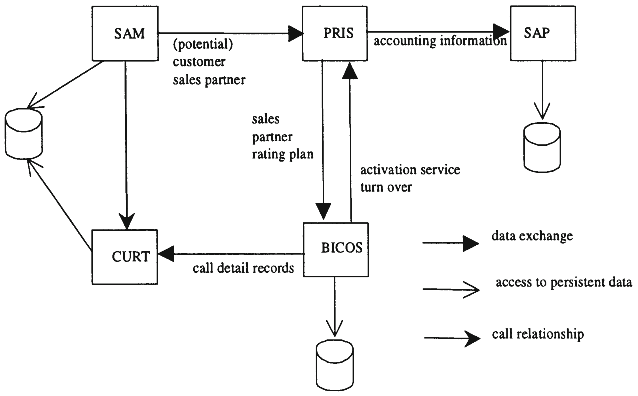

SAM
PRIS
SAP
(potential) customer sales partner
accounting information
sales partner rating plan
activation service turn over
CURT
BICOS
call detail records
data exchange
access to persistent data
call relationship

Figure 1. Initial software landscape

The applications are depicted as boxes on the upper level. The communication between the several systems is realized with services of the underlying operating system, shown at the second level. The basis for communication within a LAN/WAN is TCP/IP.

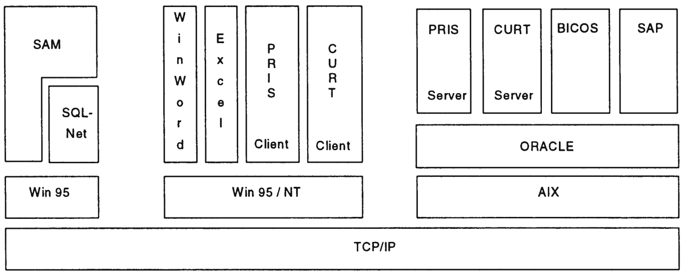


Figure 2. Technical infrastructure of software landscape

The obvious deficiencies of the software landsape (redundancies, manual data exchanges) are removed during the migration described in section 3. The result is a software landscape based on distributed objects, software components, and an integrated compository (compare section 4).

2.2 Available software systems

The software landscape contains several software systems that are briefly explained below. We discuss their main purpose, their key business objects and for a few we sketch the release policy planned.

The software systems identified in the software landscape are:

a) The **SAP/R3** modules used are SAP/FI for Finance and SAP/CO for Controlling. They deal with accounting information that is received from a provisioning system (see below). SAP/R3 impacts the vision of the future software landscape, because the structure of SAP/R3 modules and their relationships to other systems can hardly be modified. But because of the integrated data base of R3 data exchange SAP/R3 modules is no problem. Some further modules of SAP are planned to be integrated later.

b) The administration tool for customer master data **SAM** supports marketing activities (analysis and control) for sales and marketing departments. It deals with the key business objects "sales partner" and "customer data" and does the assignment between them. These business objects are also used by a provisioning system for sales partners (PRIS) (see figure 1). SAM is developed as client/server application. It uses Oracle as RDBMS, the database access is carried out with SQL-Net. The first release of SAM is dealing with "sales partner" and is running since the middle of 1997. At the end of 1998 the second release will deal with private customers as further business objects. For the next extension of SAM, WAN-wide data access is planned.

c) **PRIS** is a provisioning system for external sales partners. It creates accounting information for SAP/R3. It requires some data from SAM (business objects "sales partner" and "customer"). PRIS is client/server based like SAM, based on a 3-tier architecture. In a next version PRIS will administrate the business object "turnover". It receives these information from the billing system BICOS.

d) One further individual software system is **CURT**, an application for providing reports and statistical analysis results. It analyzes and evaluates, for example, data concerning customers' call behaviour. The report system is based on 3-tier architecture with Oracle as the underlying database.

e) **BICOS** is a billing system (derived from a standard billing system). Its key business objects are "overall turnover" and "call detail records." It has an exchange interface to PRIS. BICOS data are exported as ASCII files. Interfaces to other applications were planned at the beginning of this project.

All existing software systems have their own data repository. Some of the data exchange relationships in figure 1 represent manual exchange of data.

This requires a lot of effort and it is the reason for frequent data inconsistencies.

Besides the software systems discussed, several other systems are needed. These are either developed in-house or they are purchased. In case of purchase, the goal is to focus on standard systems as far as possible.

3. THE SOFTWARE LANDSCAPE MIGRATION PATH

The overall goal of the architecture migration was to start from the existing software architecture and to finally obtain a software architecture that is properly integrated (data integrated, user interface integrated, control integrated) and easily extensible.

After analyzing the initial landscape of the software available it turned out that the proper integration of the existing systems would affect the architecture of the software systems available substantially. In order to ensure that the existing systems remain usable while they are prepared for integration and while they are actually integrated, it was decided to subdivide the migration path into several steps. This was meant to reduce the risks of touching working systems and to focus the effort on a few systems in each step.

Another reason for this stepwise approach was that the owners of the individual software systems had different ideas about how fast their systems had to be evolved. While some wanted to stabilize their software at first (e.g., because a version had been released recently) others wanted to implement new architectural guidelines as soon as possible (e.g., because they were about to plan a new release anyway). These goals and specific ideas had to be reconciled within a common migration path.

In the following we discuss the migration steps identified and the software architectures that were achieved after carrying out these steps.

3.1 Migration step 1: data exchange support for key software systems

In order to avoid the most error-prone inconsistencies as soon as possible, we had to ensure that at least key objects were harmonized. For this purpose, we introduced the data exchange system DEXS. An example for key object harmonization were customer master data which were provided by PRIS, SAM, and SAP independently from each other. While it was not possible to eliminate this multiple responsibility for customer master data, it was at least necessary that any system that gets knowledge of customers informs all other software systems.

This simple form of data integration based on systematic data exchange yields an architecture as sketched in figure 3.

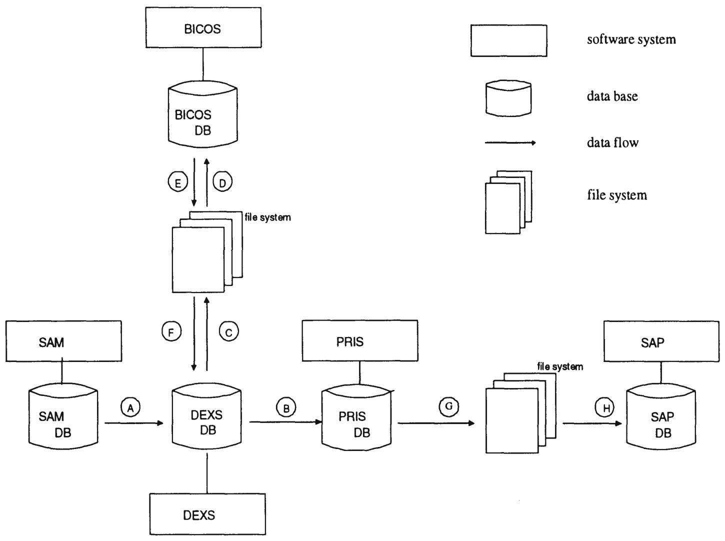


Figure 3. Exchange of business objects between software systems and DEXS

This figure shows that the redundancy of customer master data is not eliminated, but that the exchange system ensures that all customer master data identified in one system is forwarded to all other systems concerned.

business object	**master**	**data flow via interface**							
	application	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**
sales partner	SAM	x	x	x	x				
customer data	SAM	x	x	x	x				
call detail records	BICOS					x	x		
turnover	BICOS		x			x	x		
accounting information	PRIS							x	x

This first kind of integration is neither very ambitious from a software technology point of view nor does it reduce the effort that is spent for managing data within several software systems, but it ensures that all customer master data is available to all software systems concerned. This first step was quite straight-forward. After identifying the common responsibilities for certain type of data and after identifying which data was produced within which business processes and by which software systems, the most difficult task was to agree on data exchange formats between applications.

At first, the data exchange system implemented only supported the exchange of customer master data between SAM and PRIS. After eight weeks of operation it had to support a further five systems (not discussed in detail in this article). Since interfaces were defined in a bilateral way (i.e., by transforming information from the internal format of system A into the internal format of system B), the effort spent for the exchange system grew exponentially with each new system to be supported. That is why the architecture of DEXS has been changed. DEXS accepts several formats of, for example, customer master data and translates them into an internal format, that can be accessed by all software systems.

Since the internal format cannot be directly processed by the software systems, it has to be translated into each of the application-specific formats. The corresponding process supported by the data exchange component is shown in figure 4.

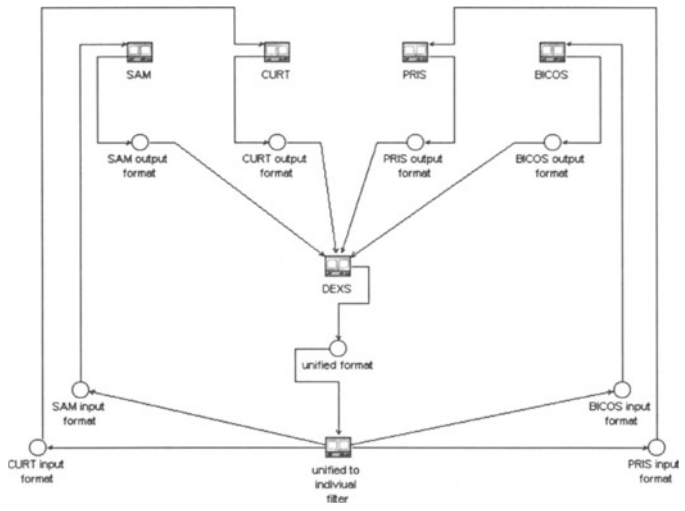


Figure 4. Data exchange system DEXS with unified exchange format

It is described in terms of FUNSOFT nets [GG95,DG98], which are high level Petri nets. Rectangular symbols represent activities, the annotations denote the software system used. Circles represent information channels, the annotations denote the type of information exchanged.

The data exchange component DEXS gets the data in an individual output format from one of the several software systems. It translates the data into a unified format, filters it into the required input format and forwards the translated data to the concerned software system.

The effort for implementing this first step was rather low. The exchange system as sketched in figure 3 did not require any modification of the software systems at all. The exchange system DEXS as sketched in figure 4 meant to provide some additional interfaces for the software systems, but their core functionality and their core structure remained unchanged. In addition, the filter "unified to individual filter" translates the unified format into the individual formats needed. Whenever a new system has to be integrated the filter has to be extended by a translation mechanism (from the unified format into the new individual format). This modification is local and does not affect the software systems themselves.

3.2 Migration step 2: data integration of key components

While the first migration step did not remove the data redundancy that was introduced by sharing the responsibility for customer master data between SAM, PRIS, SAP and CURT, the second step aimed at avoiding data redundancy at least for key applications. In terms of software architecture this means to identify common components and to delegate the responsibility for the commonly used data to them. Figure 5 shows how a common customer component is extracted from PRIS, CURT, and SAM.

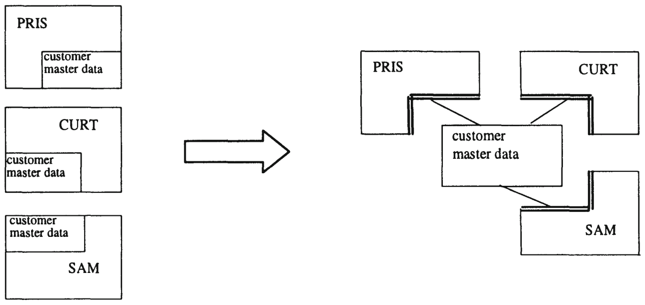


Figure 5. Exchanging common subcomponents

After splitting the software systems into two parts (key functionality and commonly used customer component), the customer component is used by all three systems, thus, ensuring that only one component cares for customer master data. According to the model-view-controller paradigm, the three components can still display customer master data in the way they were used to do. Figure 5 shows how the architecture of applications is modified by extracting a commonly used customer component.

3.3 Migration step 3: control integration of key Components

The next migration step was to improve the control integration between components that were used independently before. This control integration implemented was oriented towards the business processes needed. An example process supported by the systems SAM, CURT, PRIS, SAP systems is the process "identify new customer". Figure 6 shows an overview about this business process (again represented by a FUNSOFT net). It illustrates that first customer information is gathered by using SAM, that this information is complemented by associating the customer master data with a partner, whose provision depends on the number of customers (PRIS), that customer master data are forwarded to SAP (where debitor accounts are created), returned to SAM (where all information about the new customer is gathered) and that report information are forwarded to CURT (where the new customer is considered in the weekly "new customer report").

As soon as the business process "identify new customer" is enacted, the supporting systems have to be called in the right order and with the right parameters. In principle, this sounds like a workflow management problem. In the given situation, however, we did it without a workflow management system, but we hard-wired the business processes to be supported. The reason for doing so was that each of the available systems covers quite large chunks of the business process. Usually, workflow management based on such powerful systems leads to gross-grained process models, which are not very expressive. In our hard-wired solution, the control component (called CTRL in figure 7) enforces business processes. It knows about the processes to be supported (parts are directly implemented as control flow dependencies within CTRL, others are expressed as process parameters in the business process model) and calls SAM, CURT, PRIS and SAP accordingly. To move to a workflow based solution following the guidelines of the workflow management coalition demands to move the complete knowledge about a business process model to the business process model representation and to keep the CTRL component completely process-neutral.

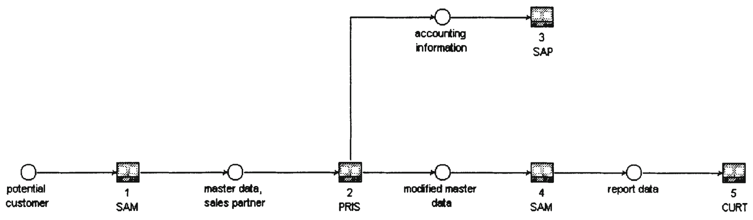


Figure 6. Business process model "Identify new customer"

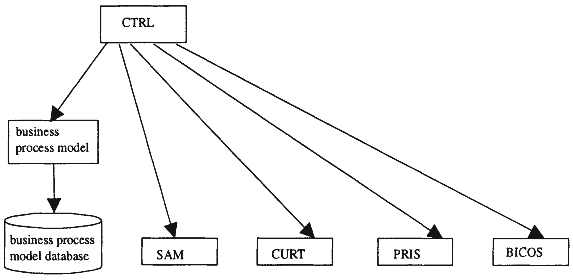


Figure 7. Integration of software systems by a control component

3.4 Migration Step 4: Implementation of Components by Distributed Objects

The German telecommunications market was deregulated recently. The business of the new telecommunication companies is not completely settled yet, but it is subject to changing market conditions and to evolving organizational circumstances. This highly flexible situation demands flexible software. Existing software has to be integrated with new software, business processes may change and have to cover more and more sites. Users to be supported work with software clients running on different platforms (ranging from PCs/laptops to workstations with different operating systems). Thus, flexibility and extensibility are key requirements for the software landscape needed. Support for various and frequently changing client platforms is a concrete requirement derived from the general requirement for flexibility.

The need for flexibility, easy assembly and support of various client platforms was accepted as the general guideline for the fourth and last migration step. Thus, we chose a component-based software architecture as our vision of the migration. In this vision, functionality is provided by components that can easily be assembled, distributed to various locations, and that help to keep the software landscape flexible and extensible. In order to provide availability on various client platforms, we decided to demand that components should be embedded into web browsers. That means, functionality offered by components can easily be accessed from all platforms supported by browsers.

Obviously, the component-based vision cannot mean replacing existing systems by componentized versions immediately because this would mean spending software development effort without gaining any new functionality. This is not acceptable in a situation in which key systems still have to be developed. Instead, the vision of a component-based software landscape serves as long-term goal towards which all software development efforts are directed. Maintenance – if necessary – should try to identify components and to extract them from existing software. New development projects should deliver software components and integration into component-based architectures should be a key requirement for any software that is purchased.

The vision of a component-based software landscape cannot serve as long-term goal if it is not made concrete. New software not only has to be component-based, it also has to match the concrete component model chosen. Thus, the vision of the software landscape has to be underpinned by an implementation vision. For implementation purposes we had to choose between the COM/ActiveX component model [Cha98], the CORBA component model [MZ95] and the Java Beans/Java Enterprise Beans component model [OH98]. It was decided to accept the CORBA component model as our basis. COM/ActiveX was denied because it is a proprietary approach, Java Beans were not accepted, because it was doubted if the Java native interface was open enough to embed the variety of available software systems that were written in various programming languages. CORBA was accepted not only because of the component model, but also because of its services that provide location transparency and convenient component naming facilities.

Another reason for the choice of CORBA was that it was rated as the best paradigm for enabling the smooth move to a component-based and Web-integrated user interface. Using the CORBA object model provided the chance to benefit from IIOP (Internet Inter-ORB Protocol) [OH98]. This protocol helps to overcome the poor interaction facilities between client and server as provided by HTTP. While HTTP is rather restricted with respect to parameter exchange and transfer of results from the client to the server, IIOP allows full exploitation of the advantages of exchanging objects between client and server. Since IIOP serves as standard for communication based on objects between clients and servers, the component-based software landscape can easily satisfy the requirement to provide the same user interfaces for all platforms supporting the usual web browsers. Figure 8 sketches the vision of such an architecture (called the ObjectWeb). It shows that the basic idea of flexible components managed by CORBA. In order to access these components and the CORBA services needed, the HTTP protocol between web browser and web server is complemented by IIOP.

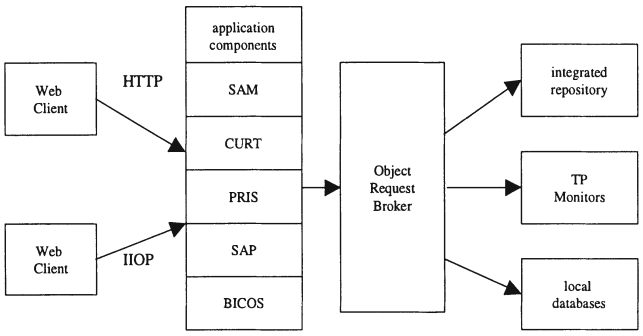


Figure 8. Vision of the ObjectWeb

The final software landscape that results from the steps mentioned avoids data redundancies, thus ensuring higher levels of consistency. It is based on a component-based architecture of software systems, thus allowing for reuse and flexible exchange of components. It is based on web browsers, thus ensuring a uniform user interface and it uses an object request broker, thereby, implementing location transparency. Even though this software landscape has not been completely reached (some software systems are not componentized yet, some cannot be executed from within a web browser, the CORBA based integration step has still to be implemented), it is an important guideline for all software development activities. Each development project can be measured against the goal to contribute to the final software landscape.

4. PROBLEMS ENCOUNTERED IN THE MIGRATION

The migration led from a software landscape that was determined by loosely integrated software systems to a tightly integrated system that is prepared for a software architecture based on a component model [Lew98] and which is homogeneously accessible from web browsers [Uma97]. At the end of the migration process it was beyond question that the software architecture had been substantially improved with respect to data consistency, data redundancy, clarity of control dependencies, extensibility and flexibility of distribution of components. Nonetheless, during migration we encountered some problems that had to be resolved. Generally speaking, these problems did not occur for the initial version of the software landscape, but they are due to the higher level of data integration, control integration and user interface integration. In the following we discuss some of these problems:

Security: While the protection against unauthorized access to data was necessary for the initial version as well as for any of the following versions, the protection against access during data transfer was an additional problem for all versions, except of the initial one. In particular, the access to software systems from within a web browser was a major obstacle to the new architecture. In order to overcome these problems we implemented a firewall with rather strict authorization checks. In addition, software systems that are accessible from the web are double-checked before released. Unless a system is released for this purpose, it runs on a completely separated local network. Thus, the production environment accessible from the web is not integrated at all with the internal local network. This, obviously, results in some problems (how to transfer a release from one local network to the web-accessible network, which criteria have to be fulfilled before a system is released for the web-accessible network), but this rather rigid approach was the only way to allow access from the web to software systems at all.

Configuration management: While the initial version (consisting of just a set of separated software systems) did not require any coordination across the borders of applications, configuration management required more effort the tighter the systems were integrated. It turned out that the role of a configuration manager had to be established. While this role was not known as an explicit one before (each leader of a development team somehow cared for his configuration issues) the coordination of release plans for various inter-dependent software systems turned out to be a full-time job. We expect that configuration management will become easier the more the vision of a truly component-based architecture is reached, because the management of component configuration requires less coordination than the configuration management for software architectures determined by the migration steps 2 and 3.

Convergence towards commonly used basis systems: The systems already available at the start of the migration, and the planned systems, used different database management systems and user interface management systems. Sometimes they were based on certain types of middleware (ranging from distributed database systems to message-oriented middleware systems). The vision of the ObjectWeb allowed details of database management systems to be abstracted away as long as they were accessible from an object broker following the CORBA architecture [MZ95] (which is true for all database management systems used). User interface issues were sorted out by demanding browser-compatible interfaces, which was accepted for all systems (even though it was not implemented immediately). Thus, the most important remaining problem was that of the middleware system used. Here, the choice of CORBA was subject to discussions, but it was finally accepted. The implementation efforts for moving systems to a CORBA-based architecture varied from system to system, but it has to be admitted, that efforts were quite substantial in a few cases (e.g., for the PRIS system, which was based on a distributed database system and on the SQLNet protocol from Oracle).

Common development plan / focusing of efforts: As long as software systems were developed independently by departments or autonomous development teams, project planning was rather easy and straightforward. With a higher level of integration additional dependencies on the release requirements of other systems, quality management plans, and management goals became obvious. We are convinced that these dependencies did exist right from the beginning, but that they were hidden. That is why we believe that it was useful to make them explicit, even though it increased the management overhead.

Common architecture model: The initially available software systems not only differed with respect to the platforms used and the ways they were subdivided into pieces, but also with respect to the architecture model chosen for describing software systems. Single teams had agreed on a vocabulary and on certain types of diagrams they used efficiently to communicate internally. They used layered models of software architecture, they used hierarchies of modules, and some used UML class and package diagrams. Some teams provided architecture information that showed how to associate software pieces with operating system processes, and explicitly defined communication protocols between operating system processes. Others just provided high-level package diagrams (which could serve as a starting point for a work breakdown structure, but not as a sufficient architecture description). A tighter integrated software landscape demanded that teams had to communicate across team borders (in order to define requirements for components they wanted to use, in order to specify interface of services they wanted to use and in order to agree on exchange formats and communication protocols between their systems). In the project we introduced a distinction between an application architecture view (data flow between functional modules, identification of key business objects), a software architecture view (modules, object types, call relationships) and a technical architecture view (distribution issues, telecommunication infrastructure needed). The use of a predefined set of architectural diagrams and the decision to use only these diagrams was cumbersome on the one hand because it meant to give up certain individual kinds of architectural descriptions. On the other hand it helped to overcome misunderstandings and useless arguments concerning the style of architecture descriptions. To sum up, the use of a common vocabulary (represented in terms of a few types of diagrams) and a clear – albeit still informal – description of its semantics turned out to be of substantial benefit with respect to a homogeneous description of the software landscape.

Management of development problems that had already started: While it was easy to ensure that new projects adhered to the software landscape and to the new architectural guidelines, it was rather difficult to ensure that project teams, which were in the middle of their projects, at least try to consider as much of landscape and guidelines as possible. In fact, we did not succeed in transferring our vision of the ObjectWeb into these kind of projects. This is all the more annoying since we spent some effort on doing so (in terms of walkthroughs and review discussions). Our conclusion from this experience is that a new vision should only be applied to new projects and that it is hardly possible to establish new architectural guidelines after project start.

5. RELATED WORK

The research in the roles and purposes of software architecture in general [SG96, PW92, Gar95], in architecture description languages [MT97, RMR98], and in patterns of software architecture [Sha95] seems to be well ahead of the industrial practice of software architecture design. While the research on software architecture and architecture description languages describes well-founded approaches to the specification of software architectures, software architectures in industrial practice seem to be determined by vague and inconsistent descriptions. Even worse, it seems as if we are rather far away from any kind of industrial de facto standard for describing software architectures. We believe that the benefits of explicit and precise architecture descriptions can at best be conveyed in the context of smooth software migrations. Here it is possible to immediately reduce costs (of extensions, maintenance and restructuring) and to improve quality of systems delivered [BJN98]. Based on these benefits it is possible to ensure the use of appropriate architecture descriptions.

The problem of describing architectures of distributed systems in industrial practice is even harder. Once again there is some research foundation [MDE95], but little consensus on how to describe distribution issues. Again, it should be demonstrated which benefits are to be gained by using homogeneous architecture descriptions.

6. CONCLUSION

We believe that the use of architecture descriptions should be embedded into a software architecture method, that explains what has to be done when and for what purpose. In the project discussed in this paper we introduced such a – still rather coarse-grained – method that is set up around the term of a software landscape. Even though this remains rather vague for the time being, we think that it allows to start where industrial projects are and to successively move towards more precise and expressive descriptions of software architectures. In particular, this not very ambitious approach helped to obtain architecture descriptions (expressed as software landscapes) that were amenable to discussions with different levels of management. Generally speaking, the approach to establish a common architectural model for the software systems of a telecommunication company demands the careful definition of a migration path that compromises between two goals:

1. fast migration to a state-of-the-art software architecture
2. smooth migration that ensures 100% availability of working systems

The compromise between these two goals must ensure that any effort spent on evolving or replacing software systems must bring the software architecture closer to the vision of the final software landscape.

We believe that our experience is not only typical with respect to the intermediate migration steps taken, but also with respect to the final vision of our migration. The current trends towards component-based software development on the one hand and towards components being executed under the control of a web browser on the other hand naturally lead to the vision of the ObjectWeb.

REFERENCES

[BJN98] K. Bohrer, V. Johnson, A. Nilsson, B. Rubin (1998). “Business Process Components for Distributed Object Applications” *Communications of the ACM* 41(6), 43-48.

[Cha96] D. Chappell (1996). *Understanding ActiveX and OLE* Redmond, Washington, US: Microsoft Press.

[DG98] W. Deiters, V. Gruhn (1998). “Process Management in Practice - Applying the FUNSOFT Net Approach to Large Scale Processes” *Special Issue on Process Technology / Automated Software Engineering* 5, 7-25.

[Gar95] D. Garlan (1995). “First International Workshop on Architectures of Software Systems: Workshop Summary” *ACM SIGSOFT Software Engineering Notes* 20.

[GG95] G. Graw, V. Gruhn (1995). “Process Management in-the-Many” *Software Process Technology - Proceedings of the 4th European Software Process Modeling Workshop*, W. Schäfer (editor) Noordwijkerhout, Netherlands: Springer 163-178. appeared as Lecture Notes in Computer Science 913.

[Lew98] S.M. Lewandowski (1998). “Frameworks for Component-Based Client/Server Computing” *ACM Computing Surveys* 30(1), 3-27.

[MDE95] J.Magee, N. Dulay, S. Eisenbach, J.Kramer (1995). *Specifying Distributed Software Architectures* European Software Engineering Conference, Barcelona, Spain, 137-153, appeared as LNCS 989.

[MT97] N. Medvidovic, R.N. Taylor (1997*). A Framework for Classifying Architecture Description Languages* European Software Engineering Conference, Zurich, Switzerland, 60-76, appeared as LNCS 1301.

[MZ95] T.J. Mowbray, R. Zahavi (1995). *The Essential CORBA: Systems Integration Using Distributed Objects* Toronto, Canada: Wiley.

[OH98] R. Orfali, D. Harkey (1998). *Client/Server Programming with JAVA and CORBA* Toronto, Canada: Wiley.

[PW92] D.E. Perry, A.L. Wolf (1992). “Foundations for the Study of Software Architectures” *ACM SIGSOFT Software Engineering Notes* 17(4).

[RMR98] J.E. Robbins, N. Medvidovic, D.F. Redmiles, D.S. Rosenblum (1998). *Integrating Architecture Description Languages with a Standard Design Method* International Conference on Software Engineering, Kyoto, Japan.

[SG96] M. Shaw, D. Garlan (1996). *Software Architecture: Perspectives on an Emerging Discipline* Prentice-Hall.

[Sha95] M. Shaw (1995*). Patterns of Software Architectures, in: Pattern Languages of Program Design* edited by J.O. Coplien and D.C. Schmidt, Addison-Wesley.

[Uma97] A. Umar (1997). *Object-Oriented Client/Server Internet Environments* New Jersey, US: Prentice-Hall PTR.

Structural Analysis of the Software Architecture - A Maintenance Assessment Case Study

Catherine Blake Jaktman, John Leaney, and Ming Liu
Computer Systems Engineering, Faculty of Engineering, University of Technology, Sydney, Australia
{cjaktman, jleaney, mliu}@eng.uts.edu.au

Key words: Software architecture, software evolution, maintenance measurements, experience report, architectural erosion.

Abstract: Architectural erosion is a sign of reduced architectural quality. Quality characteristics of an architecture, such as its ability to accommodate change, are critical for an evolving product. The structure of an architecture is said to be eroded when the software within the architecture becomes resistant to change or changes become risky and time consuming. The objective of our work is to understand the signs of architectural erosion that contribute to decreased maintainability. A maintenance assessment case study is described in which we apply structural measurements to a product to determine signs of architectural erosion. It provides an understanding of a product's quality by examining the structure of its architecture. The ability to assess architectural erosion in an evolving software product allows the quality of the architecture to be monitored to ensure its business and maintenance goals are achieved.

1. INTRODUCTION

1.1 Software evolution

Successful software systems experience continual evolution (Lehman 1989) due to events in the system's environment, usage, and business domain. Software systems frequently experience increased maintenance (Parnas 1994) and degradation from continual changes made during software maintenance activities (Bohner 1991). Often, once a system begins to show signs of degradation, the maintainability of the system will continue to worsen over time as more software changes are implemented in a system that has grown in size and complexity.

During software evolution, the high-level organisation of the overall system, the software architecture, becomes the critical aspect of design (Garlan 1995; Grisworld and Notkin 1995). The software architecture represents the organisation of a system as a composition of components, connections, and constraints (Garlan 1995). The structure of the architecture includes the gross organisation of the system and global control structure; protocols for communication, synchronisation, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives (Garlan and Shaw 1993). The structural properties of a software architecture can be expressed in terms of components, interrelationships, principles, and guidelines about their use.

1.2 Architectural quality

When the quality of the architecture deteriorates functional adaptations become difficult (Bakker and Hirdes 1995; Kogut and Clements 1998). Additionally, design decisions made at the architectural level have far-reaching consequences on the resultant code (Turver and Munro 1994). Certain architectural design decisions may restrict the ability of a software component or interface to be easily modified, or require many components to be modified. Architectural quality is also important for organisations that have evolving product-line architectures. A well-executed and maintained architecture enables organisations to respond quickly to a redefined mission or new and changing markets (Dikel et al. 1997; Morris and Ferguson 1993).

The focus of our work is to understand the signs of reduced architectural quality leading to an increase in maintenance difficulty during the evolution of a product. We want to be able to monitor the maintainability of a product throughout its evolution, to allow functional and non-functional software requirements to be implemented without affecting the following factors

- the flexibility or extendibility of the product
- the understanding of the software architecture
- the maintenance effort required to perform maintenance tasks

1.3 Architectural erosion

Our interest in architectural quality is similar to work in architectural erosion and drift by Perry and Wolf (Perry and Wolf 1992). Perry and Wolf define architectural erosion as "*violations in the architecture that lead to* *increased system problems and brittleness*". They define architectural drift as "*a lack of coherence and clarity of form which may lead to architectural violation and increased inadaptability of the architecture*". Their work mentions the importance of the architecture's style to encapsulate decisions about the architectural elements, constraints and relationships that are needed to understand architectural violations such as erosion and drift.

We have extended Perry and Wolf's definition of erosion to include the structure of an architecture. We define the structure of a software architecture to be *eroded* when the software within the architecture becomes resistant to change or software changes become risky and time consuming. Erosion can also be exhibited when the software is hard to understand or manage due to an increase in the size and complexity of the code and its structure. Erosion can be a result of poor design decisions made whilst implementing maintenance changes to the system, or a result of limited architectural understanding during software maintenance that may have constrained the flexibility of the design.

1.4 Characteristics of erosion

The characteristics of architectural erosion in an evolving product are listed below in *Table 1*.

Table 1: Characteristics of erosion

The complexity of the architecture has increased from a previous release as shown by an increase in the structural complexity measurements.
The impact of a software modification results in unpredictable software behaviour (e.g., ripple effect).
The architecture is not documented or its structure is not explicitly known.
The relationship between the architectural representation and the code is unclear or hard to understand.
There is a continual increase in the defect rate that is disproportionate to the amount or type of maintenance work performed (e.g., new functionality added or technology upgrades).
Greater resources are required to implement a software change (i.e. understand, code and test).
Experience with the software becomes crucial to understanding how to implement a software change.
Certain software changes may become too risky or costly to implement.
The design principles of the architecture are violated when implementing a product variant (e.g., code redundancy due to cloning).
The system may become resistant to change (i.e. "brittle"), or require additional operational procedures (e.g., manual tasks) to support new functionality.

1.5 Structural signs of erosion

The signs of architectural erosion are determined by studying the changes in the size and complexity of the product's architecture. If the results of the measurements change in such a way to indicate increased maintainability (i.e., increased complexity), the architecture is studied further to determine if the measurements indicate the presence of erosion. The signs of erosion are validated by the software developers to ensure agreement that the software has eroded. In the long term this may be developed as a sensitive indicator of impending erosion, and the architectural quality preserved.

A description of the architectural measurements we have chosen to help us understand architectural erosion are included in sections 2.4, 2.5, and 2.6 of this paper. The measurements were chosen on the basis of their history of usage and understanding, and their ready availability.

2. MAINTENANCE ASSESSMENT CASE STUDY

This section describes the framework of the maintenance assessment case study. The objective of the maintenance assessment case study is to identify useful measurements that will allow us to determine the signs of architectural erosion in an evolving product.

2.1 Approach

In the maintenance assessment case study we apply structural measurements to the architectural properties of a product. The measurements are applied to each operational release of the product throughout the evolution of the product. They are generated using the Logiscope™ code analysis tool (Logiscope 1997-1998). The measurements between each release are compared and analysed to detect signs of architectural erosion. The steps in the maintenance assessment approach are shown in *Table 2*.

Table 2: Case study process

Step 1: Select an architectural viewpoint for analysis of the system. • Criterion: the architectural viewpoint should conform to some accepted viewpoint
Step 2: Take measures on selected releases of the product. • Criterion: structural measures selected should conform to recognised measures • Criterion: selected measures should be fair indicators of structural erosion as described in sections 1.3 Architectural erosion and 1.4 Characteristics of erosion • Criterion: the selected releases should give a fair view of the product

Step 3: Analyse the structural measures for change. • Criterion: change should be such as to be attributable to erosion • Criterion: change should be defined as outside some expected 'noise' values
Step 4: Interpret the measures in terms of (structural) erosion. • Criterion: interpretation should be consistent with structural erosion as described in sections 1.3 Architectural erosion and 1.4 Characteristics of erosion
Step 5: Translate the structural erosion conclusions into (maintenance) programming terms. • Criterion: use documented maintenance programming practice terms
Step 6: Validate with project maintenance programmers (interviewees) • Criterion: ensure the experiment is blind, i.e. the interviewees do not know of the analysis before the interpretation is complete. • Criterion: ensure terms are fully understood by interviewees • Criterion: use accepted techniques such as questionnaires • Criterion: ensure interviewees have 'last word' on conclusions

2.2 Proposed measures

The proposed measures used to determine architectural erosion include:
- general measures
- basic architectural measures
- derived architectural measures

2.3 General measures

The general measures listed in *Table 3* are taken for each selected software release of the product. The general product measures are used to understand the age and growth in the size of the product.

Table 3: General measures

Version No.
Date of Release
Lines of Code
No. of Runtime Files
Total Components

2.4 Basic architectural measures

Architectural measures are taken to provide an understanding of the overall view of the system. Architectural measures are based on the call graph representation of the product and represent the complexity of the structure of the architecture. The basic architectural measures (or counts) are defined in *Table 4*.

Table 4. Basic architectural measures

Components (or Nodes) Number of Components (or Nodes)	Number of components (or nodes) in the relative call graph. An architectural-level component represents a source code file.
Edges Number of Edges	Number of calls between components. This is the same as edges in the relative call graph. A relative call graph is a diagram that identifies the components in a system and shows which modules call one another.
Call Paths Number of Relative Call Graph Call-Paths	Number of calling paths in the relative call graph, from the component to each leaf components in the graph.
Levels Number of Relative Call Graph Levels	Number of hierarchical levels in the relative call graph.

2.5 Derived architectural measures

The *derived* measurements taken at the architecture-level (based on the "counts" of *Table 4*) are defined and explained in *Table 5*. The measurements listed in *Table 5* are supported in the Logiscope™ code analysis tool (Logiscope 1993).

Table 5: Derived architectural measures

Hierarchical Complexity Relative Call Graph Hierarchical Complexity	Average number of components per relative call graph level (number of components divided by number of levels).
Structural Complexity Relative Call Graph Complexity	Average number of calls per component (number of calls between components divided by number of components).
Average Components/Path	Average number of components per call path (components divided by call paths)
Average Paths/Component	Average number of paths per component (call paths divided by the number of components.)

3. THE CASE STUDY

3.1 The Squid product

In the case study we applied the aforementioned measurements to the Squid product (Squid), which was developed at the National Laboratory for Advanced Networking Research (NLANR) at the University of California at San Diego. Squid is an Internet object caching product. The Squid product was chosen as historical data was publicly available for each operational release of the product throughout its evolution. The Squid product is written in the C programming language and is implemented in the Unix operating environment. Six releases of Squid spaced approximately at equal time intervals were chosen for analysis.

3.2 Approach

In our approach structural measurements are taken for each release of the Squid product. We then interpret the measurements, and put the interpretations to the Squid developers to see if our interpretation of the measures made any sense to them. (Being our initial case study, there were no hypotheses developed. The purpose of this work was to see if we could find any encouragement for using existing structural measurements for this research problem.) The steps followed in the case study are outlined in *Table 2*; the sub-headings below follow the steps outlined in the table.

3.2.1 Step 1: Select architectural viewpoint

The ideal architectural viewpoint would be to work with the subsystems defined by the original software developers, and understand changes in those subsystems and the relationships between them. This would be in accord with the software architecture definition whereby one represents the organisation of a system as a composition of components, connections and constraints (Garlan 1995).

From this perspective, Squid consists of the following major sub-systems:

- the Client Side (icp.c, client_side.c)
- the Server Side (proto.c, http.c, ftp.c, gopher.c, wais,c, ssl.c, pass.c)
- the Storage Manager (store.c)
- other sub-systems, including Neighbours; IP/FQDN Cache; DNS Servers; Cache Manager; Network Probe Database; Redirectors and Access Controls.

Unfortunately, to quote the Squid developers, "the Squid source code has evolved more from empirical observation and development rather than a solid design process over two years or more. It carries a legacy of being 'touched' by numerous individuals, each with somewhat different techniques and terminology" (Wessels 1997).

We have created an architectural view based on "the view from the top". The architecture viewpoint, which appears repeatable for procedural languages such as C, is to choose the 'main' routine as a reference node for structural measures. Our structural measures are based on the call graph, and are taken from the main routine alone. Thus, we will only include components that are called on a path from main.

3.2.2 Step 2: Take measures of selected releases

In this step the general, architectural, and derived architectural measures are taken on the selected releases of the product.

3.2.2.1 Squid - general measures

The general measures for Squid, as specified in section *2.3 General measures*, are given in *Table 6* below.

Table 6: Squid general measurements

Version No.	V1.0.0	V1.0.22	V1.1.0	V1.1.10	V1.1.14	V1.1.21
Date of Release	Jul. 1996	Oct. 1996	Dec. 1996	Apr. 1996	Jul. 1997	Apr. 1998
No. of Files	32	33	42	44	44	44
Lines of C Code	23587	24996	27425	30008	30222	29329
Total Components	619	630	693	733	739	748

3.2.2.2 Squid - architectural measures

The architectural measures for Squid, as specified in *2.4 Basic architectural measures* are given in *Table 7* below. The viewpoint is from the main routine. The reduction in the number of nodes and edges accords with the reduction in the lines of code in *Table 6*..

Table 7: Squid basic architectural measures (counts)

Version No.	V1.0.0.	V1.0.22	V1.1.0	V1.1.10	V1.1.14	V1.1.21
Levels	18	21	23	19	21	20
Nodes	413	421	416	429	438	434
Edges	1243	1267	1319	1399	1428	1415
Call Paths	24264	28037	32784	40616	80848	80952

3.2.2.3 Squid - derived architectural measures

The derived architectural measures for Squid, as specified in *2.6 Derived Basic architectural measures* are specified in *Table 8* below.

Table 8: Squid derived architectural measures

Version No.	V1.0.0	V1.0.22	V1.1.0	V1.1.10	V1.1.14	V1.1.21
Call Paths	24264	28037	32784	40616	80848	80952
Levels	18	21	23	19	21	20
Hierarchical Complexity	22.94	20.04	18.08	22.57	20.85	21.70
Structural Complexity	3.01	3.01	3.17	3.26	3.26	3.26
Average Paths/Component	58.75	66.59	78.8	94.67	184.58	186.52
Average Components/Path	0.01	0.01	0.01	0.01	0	0

3.2.3 Step 3: Analyse the structural measures for change

The major changes that occurred, and the discussion, are shown in *Table 9*. Other changes were considered to be too dubious to discuss. Considering all the data presented to date, a strong observation can be made that the system started to stabilise at V1.1.0. This is shown by a slower growth rate in the size of the product and a reduced rise in the complexity of the product. Therefore, we will only comment on changes after V1.1.0.

Table 9: Analysis of changes in Squid

Property	Discussion
Components	The number of components has been practically constant throughout the life of the project.
Call paths	The number of call paths is arguably high compared with recommendations in (Logiscope 1993) even from the beginning of the project. The number of call paths has changed by a factor of (about) 3.5 during the life of the project, with a sharp rise between V1.1.10 and V1.1.14 (as shown in Table 7).
Component maintainability	The components were analysed using the ISO 9126 maintainability measures (ISO 1991). The component maintainability for Squid was consistent thought the product evolution (i.e. 2% excellent, 89% good, 4% fair, 3% poor and 2% undefined).

3.2.4 Steps 4 and 5: Interpretation and translation

These steps involve interpreting the measures in terms of (structural) erosion and translating the structural erosion conclusions into (maintenance) programming terms. The analysis of changes in Squid *(Table 9)* suggests that with the increase in call paths, and the number of components remaining the same, that the changed (or new) components will have become much harder to integrate with each release. However, the small change to component maintainability suggests that changes to the component itself will not be hard.

3.2.5 Step 6: Validate with project maintenance programmers (interviewees).

A basic questionnaire was developed based on the conclusions of *Step 4: Interpret the measures in terms of (structural) erosion, and Step 5: Translate the structural erosion conclusions into (maintenance) programming terms.* This was sent (via e-mail) to the Squid software team leader, in order to get a response from a maintenance programmer. The reply was then analysed, and interpreted back as a second series of questions. A final conclusion was then drawn.

Based on this interpretation, the following statement was put to the Squid programmers:

> "Our conclusion would be that individual modules in Squid are OK to maintain, but integration of the modules into the Squid system can at times be difficult."

The first response to this statement included a lack of match between the statement and the Squid system. This is exemplified by the two responses, "I do not think there is a good notion of modularity in Squid." and "Thus, I cannot answer your question since I do not see a lot of well-defined "modules" in Squid." In the light of these responses, we sent another set of questions:

Our statement: Typically, the extra load of a module (doing Y and Z as well as X in your email) occurs as a result of change. Side effects can also occur as a result of change, or simply wishing to get the job done ASAP.

Squid response: This is certainly true, but, from what I know, certain "extra load" was there from the very beginning of Squid. Also note that HTTP itself ..... often forces programmers to violate many good software engineering design principles.

Our statement: You attempt to modify some part of Squid. As a result you find that a particular collection of code (an approximate module) is the most likely place to start.

Squid response: OK. Although any serious change would require modification in several files/modules.

Our statement: You then find that a piece of code does other things as well, so you have to be careful to change only what you want to change. And then there is the question of side effects. So I relate that to being able to change parts of code, but finding it more difficult to get Squid running properly again.

Squid response: I somewhat agree with your last statement. You see, when I modify a part of the code, I usually know *a priori* that I will have to modify other places. Thus, if somebody asks me a question like "is it easy to add feature W?", I reply based on the *total* amount of modifications that I foresee, and not based on the first change that I will make. In other words, most significant changes will require modification of several "modules", which is hard. And nobody cares about minor updates, I guess.

3.3 Conclusion from the interviews

If we now return to our original statement, there seems to be considerable reinforcement for the proposition that integration of modules into Squid is difficult. But, it is not obvious that individual modules are easy to maintain from the interview discussion. We can determine that the *component* level maintenance measures (ISO 9126) will not reflect the increased complexity that is shown at the architectural level (as indicated by the call path measure).

The problem of software integration is due to multiple functional modules and side effects. This is supported by the increase in call paths, with little increase in the number of components. The problem of side effects could possibly have been seen by the decrease in the number of comments, as the system became older and more difficult to understand.

3.4 Summary and conclusions

In *Table 10*, a summary of the characteristics of erosion are matched with the measurements and interviews from the Squid software team.

There is some promise in using structural measures to predict erosion. This has been an exploratory case study, but nevertheless the ability to predict some of the features of the Squid system are most encouraging.

Table 10: Characteristics of erosion matched with measures and interviews

Characteristic	Measures	Interviews
The complexity of the architecture has increased from a previous release as shown by an increase in the structural complexity measurements.	Shown in structural measures (call paths).	Not validated, we don't have coverage of enough releases, yet.
The impact of a software modification results in unpredictable software behaviour (e.g., ripple effect).	Shown in structural measures (call paths).	Validated
The architecture is not documented or its structure is not explicitly known.	No documentation of the architecture was available.	Validated
The relationship between the architectural representation and the code is unclear or hard to understand.	No documentation of the architecture was available.	Validated
There is a continual increase in the defect rate that is disproportionate to the amount or type of maintenance work performed (e.g., new functionality added or technology upgrade).	Unknown defect rate data not captured.	Not validated
Greater resources are required to implement a software change (i.e. understand, code and test).	Unknown data not captured.	Not validated
Experience with the software becomes crucial to understanding how to implement a software change.	Suggested by structural measures (call paths).	Validated
Certain software changes may become too risky or costly to make.	Unknown data not captured.	Not validated
The design principles of the architecture are violated when implementing a product variant (e.g., code redundancy due to cloning).	Unknown data not captured, although can be determine based on changes in architectural views and component usage.	Not validated
The system may become resistant to change (i.e. "brittle") or requires additional operational procedures (e.g., manual tasks) to support new functionality.	Unknown data not captured. May not be displayed (yet) in a system that is only 2 years old.	Not validated

Our study showed that in order to understand the quality of the architecture one must look at the component measures, as well as the structural architectural measures, in order to ascertain the origin of change in the structure of the architecture. For example, changes to the structure could occur as a result of a redesign that gives rise to the addition of components, thus changing the connections. Alternatively, modifications to components can affect the structure through additional calls to other components, and side effects. In the case of the Squid system, very little happened to the structure through redesign, as shown by the relatively constant number of components. However, the modules were changed to give new functionality (as described in the interviews), this is shown by the increase in the number of call paths. However, it is interesting that the number of call paths was high from the beginning, suggesting an early flaw in the architecture.

4. RELATED WORK

The most extensive software evolution study has been the FEAST project which started in 1997 (Lehman et al. 1997). This work and subsequent work on the FEAST project formed the foundation for describing the laws of software evolution and the properties of a E-type system. In this work the size of modules were used to understand the growth rate and change rate of the system, and its affect on software quality. Additional software evolution work that applied metrics included projects to determine; code degradation (Ash et al. 1994), entropy (Coleman 1995; Harrison and Cook 1990) and erosion (Kogut and Clements 1994). The focus of such work is at the program code level, using measurements such as LOC, Halstead, and McCabe, to determine the complexity of the program that may be error-prone or change-prone.

However, quantitative approaches to understanding architectural quality have been limited. The evolution of a telecommunications system has been studied to identify modules or sub-systems in the architecture that should be considered for restructuring or re-engineering (Gall et al. 1997). In their work they considered the size of each system, the change rate and growth rate. The complexity of the architecture has also been measured using pattern coverage techniques: the proportion of an architecture that can be covered by patterns and the number of patterns it takes to cover the architecture (Kazman and Burth 1998). These are complementary measures of the system's regularity, and hence its architectural complexity. These quantitative approaches differ from our work as their focus in on identifying weak areas of the architecture for re-engineering. Our work uses structural measurements to determine the maintainability of the architecture to support the evolution of the product.

Qualitative techniques such as the Software Architecture Analysis Method (SAAM) (Kazman et al. 1994), and the AQA (Architecture Quality Assessment) (Hilliard, Kurland, and Litvintchouk 1997) provide a method to gain information about a system's qualities (e.g., modifiability, security, portability). Such techniques can be used for multiple purposes such as, to consider future changes to the system or how an architecture will accommodate change. However, the qualitative architecture evaluation techniques do not clearly state a criterion for maintainability that can be measured and verified. Additionally, architectural evaluation techniques are limited in providing an understanding of the signs and causes of reduced quality.

5. BENEFITS

The maintenance assessment case study provides preliminary work in determining a set of measures that can be applied to understand the quality of a software architecture during its evolution. The ability to identify signs of architectural erosion allows us to make improvements to the architecture to increase its flexibility and longevity prior to reaching further erosion. An architectural maintenance assessment method can also be used to

– derive quality benchmarks that are necessary to evolve the product; and
– build knowledge about the factors that influence the quality of the product-line architecture.

This work can also provide insights into the causes of erosion, allowing us to improve architectural analysis and design techniques in the early phases of system development. For example, proven guidelines for assessing the extendibility and flexibility of an architecture will allow us to assess the adaptability of an architecture using pre-design data (e.g., module hierarchy).

6. FUTURE WORK

We would like to continue our research with the Squid product to gather measurements on additional software releases of the system as the product evolves. Additionally, we would like to apply this research to a system that is older and larger; this would allow us to study differences in the results that may be due to varying organisational, software process, and architectural design factors.

REFERENCES

Ash, D., Alderete, J., Yao, L., Oman, P. W., and Lowther, B. 1994. "Using Software Maintainability Models to Track Code Health", *Proceedings of the 1994 International Conference on Software Maintenance.* pp. 154-160.

Bakker, G., and Hirdes, F. 1995. "Recent Industrial Experiences with Software Product Metrics". *Proceedings of the Second Symposium on Software Quality Techniques and Acquisition Criteria.* Florence, Italy. pp. 179-191.

Bohner, S. A. 1991. "Software Change Impact Analysis for Design Evolution", *Proceedings of the 8th Int'l, Conference on Software Maintenance and Re-engineering*. IEEE CS Press., Los Alamitos. pp. 292-301.

Coleman, D., Ash, D., Lowther, B., and Oman, P. 1995. "The Application of Software Maintainability Models in Industrial Software Systems". *Journal of Systems and Software.* **29**:3-16.

Dikel, D., Kane, D., Ornburn, S., Loftus, B., and Wilson, J. 1997. "Applying Software Product-Line Architecture". *IEEE Software.* **30**:49-55.

Gall, H., Jazayeri, M., Klösch, R., and Trausmuth, G. 1997. "Software Evolution Observations Based on Product Release History", *Proceedings of the International Conference on Software Maintenance*, Bari, Italy, October. pp. 160-166.

Garlan, D. 1995. "First International Workshop on Architectures of Software Systems Workshop Summary". *ACM SIGSOFT, Software Engineering Notes.* **20**:3:84-89.

Garlan, D., and Shaw, M. 1993. An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering, Vol I, River Edge, NJ:World Scientific Publishing Company.

Grisworld, W. G., and Notkin, D. 1995. "Introduction to the Special Issue on Software Architecture", *IEEE Transactions on Software Engineering*. (April) **21:**9-287.

Harrison, W., and Cook, C. 1990. "Insights on Improving the Maintenance Process Through Software Measurement". *Proceedings of the Conference on Software Maintenance.* San Diego, CA., pp. 37-45.

Hilliard III, R., F., Kurland, M. J., and Litvintchouk, S. D. 1997. Mitre's Architecture Quality Assessment. *Software Engineering & Economics Conference.*

ISO 1991. International Standard ISO/IEC 9126. Information Technology Software Product Evaluation – Quality characteristics and guide-lines for their use, International Organization for Standardization, International Electrotechnical Commission, Geneva.

Kazman, R., Bass, L., Abowd, G., and Webb, M. 1994. "SAAM: A Method for Analyzing the Properties of Software Architectures". *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy.

Kazman, R., and Burth, M. 1998. "Assessing Architectural Complexity". *Proceedings of the 2nd Euromicro Working Conference on Software Maintenance and Re-engineering (CSMR'98).* IEEE Computer Society Press.

Kogut, P., and Clements, P. 1994. "The Software Architecture Renaissance", *Crosstalk, The Journal of Defense Software Engineering.***7**:20-24.

Lanubile, F., and Visaggio, G. 1992. "Software Maintenance by Using Quality Levels", *Workshop on Software Quality: Measurement and Practice.*

Lehman, M. M. 1989. "Uncertainty in Computer Application and its Control Through the Engineering of Software". *Software Maintenance: Research and Practice.* **1**:3-27.

Lehman, M.M., Ramil, J. F., Wernick P.D., Perry, D. E., and Turski, W. M. 1997. "Metrics and the Laws of Software Evolution - the Nineties View". *Proceedings of the Fourth Software Metrics Symposium.* pp. 20- 32.

Logiscope™ Editor Manual, 1993.

Logiscope™ by Verilog, Version 2.1, 1997-1998. Supported by Prophecy Technology in Australia, URL: http://www.prophecy.com.au

Morris, C. R., and Ferguson, C. H. 1993. "How Architecture Wins Technology Wars", *Harvard Business Review*. pp. 86-96.

Parnas, D. L. 1994. "Software Aging. *In Proceedings of 16th International Conference on Software Engineering (ICSE 16)*, Sorrento, Italy. pp. 279-298.

Perry, D. E., and Wolf, A. L. 1992. "Foundations for the Study of Software Architecture". *ACM SIGSOFT Software Engineering Notes*. **17**:4:40-52.

Squid Internet Object Cache, Source code distributions URL: http://squid.nlanr.net/

Turver, R. .J., and Munro, M. 1994. "An Early Impact Analysis Technique for Software Maintenance". *Journal of Software Maintenance: Research and Practice.* **6**:35-52.

Wessels, D. and Squid developers, 1997. Squid Programmers Guide (Draft).

Architectural Evolution

Nokia Mobile Phone Case

Juha Kuusela
Nokia Research Center, Helsinki, Finland
juha.kuusela@research.nokia.com

Key words: Architectural evolution, product family, and organization.

Abstract: Similar software products can be developed as a product family. Common architecture, addressing all common requirements of products in the family, provides the basis for wide scale reuse within the family. When independent products continue their evolution, they face new requirements that may prove to have wider scope and need addressing at the family level. However, changes on the family level may be very costly for the product projects. Our experience shows that architectural evolution is possible and practical if each change has been carefully planned, taking into account its organizational aspects. Then the change has to be carried out so that the product line does not stop. Large architectural changes are high-risk operations; even when they succeed, they tend to take much longer than expected.

1. INTRODUCTION

The software architecture group at Nokia Research Center was established in 1994. Our group has now 17 members and operates both in Helsinki and in Boston. We lead two international research projects ARES and FAMOOS within the Esprit program but our main task is to support Nokia business units in their product development. With the business units we analyze, assess and model their product architectures and give suggestions on how to improve them. We also participate in developing architecture for new product concepts.

In the cooperation with Nokia Mobile Phones (NMP), our role has been to facilitate the process, introduce state of the art in architectural design and description to software architects from NMP, review and comment their designs. The final architectural choices and their implementation have always been the responsibility of NMP architects. During these 4 years, we have had an opportunity to observe how software evolves in response to changing requirements and to learn how this evolution affects a software development organization and its development process. This experience report gives an overview of this process, presents examples of architectural evolution, and offers a classification of different architectural changes and an observation on how difficult they are to implement.

Nokia Mobile Phones produces a range of similar mobile phones. It has an opportunity to control the properties and quality, and to reduce the development, maintenance, support, and marketing costs of each product by sharing some of the effort and parts between these phones. In order to manage such sharing, the phones are organized into a product family. There are many reasons for variation in the mobile phone family. Different market segments have different characteristics and the products must offer a choice of functional features and capabilities to satisfy a wide spectrum of customer requirements. National standards often impose constraints on product functionality. Cultural differences and fashion add variation to the user interface design. Advances in technology require frequent migration of products to new platforms and environments.

Software architecture provides the basis for reuse within the product family but it also ties the products together and limits their evolution potential. Architecture can only be designed to accommodate anticipated variation. Some of the reasons for variation in the mobile phone family are rather stable (like different languages) but most are volatile and can only be anticipated few years ahead. Once the architecture can no longer support the product family, it has to be changed.

Architectural changes can be very costly. Much work is needed to update everything and changes in the basic premises may force redesign of large parts of the system. There are many sources of architectural changes; some changes can be avoided by careful planning but others are unavoidable. If the products based on the architecture are successful, new products with new properties will be added to the family. Architecture has to be periodically updated to support the new needs. This paper summarizes the architectural evolution of Nokia mobile phone product family.

2. NOKIA MOBILE PHONE FAMILY

Traditionally a cellular phone consists of a transmitter and a receiver for communication with the network, a user interface consisting of a keyboard and a display, a battery, a microphone, and a speaker. In addition, the phone has a processor and memory for the software needed for controlling the hardware. The phone may also have facilities for some auxiliary services, such as data communication.

A cellular phone communicates through the cellular network. Nokia has developed phones for various network standards, e.g., for analogue standards NMT, AMPS and TACS, and for digital standards such as the Japanese JDC and the European GSM. TDMA and CDMA are adopted in North America. For each cellular standard, Nokia provides several phones for different market segments. These phones vary in style, functionality and price. The variation is implemented both in the hardware and in the software. The development organization is large and globally distributed.

The complexity of the product family, the structure of the development organization, and the need to introduce new features as they become available in the networks, makes mobile phone software development a challenging task. Hardware development is the basis for competition but in order to benefit from this potential, new phones have to be on the market before the competitors' models using the same hardware. Software development time costs money.

So far, the evolution of the mobile phone has had only a few basic drivers. *Miniaturization* has shrunk the size from portable (like a suitcase) to devices weighting less that 100 grams. At the same time *operation-time* with a standard battery has grown from hours to weeks and *price* has dropped from the status-symbol level to that of an affordable personal phones for each family member. Through this evolution, the product concept has been rather stable: mobile phones are used for voice communication. Now we also see evolution of the product concept. The phone has an increasing role as a portable terminal to an information system and as a communication device between information systems. Intelligent add-ons see the phone as a center of a distributed system, car electronics as a general-purpose communication device, and Internet based systems as a portable browser. This change in the product concept has a large impact on the product structure.

3. INITIAL ARCHITECTURE AND DEVELOPMENT PROCESS

Initially, the software architecture of the phone addressed only the basic requirements variation in the hardware, the communication standards, and the user interface. These domain characteristics formed the basis for the initial module architecture and this architecture had only three subsystems:

1. A cellular subsystem for managing the connection to the network.
2. An application subsystem that includes the user interface software.

3. A device subsystem for interfacing with the hardware.

The separation of the cellular subsystem is critical since it allows easy development of different phones for each network and similar phones for different networks. Separation of the device subsystem is also crucial to be able to benefit from constant hardware evolution.

The development process for this architecture was very product centered. The development organization would base each new project on some earlier version of the subsystems and make the necessary modifications. This allowed each development organization to be rather independent supporting the rapid growth of a distributed organization.

4. TENSION IN THE INITIAL ARCHITECTURE

The initial architecture and the development process were very successful. However, the architecture had many inherent problems. The very separation of application software from cellular subsystem creates a problem. Different network standards have different capabilities and thus application software is in reality coupled with the cellular subsystem. The coupling is visible in the specifications since some user interface applications involve complex protocols and their specification is included in the protocol standards.

The subsystems are too large. Divide and conquer does not really work well if you only divide by three. Naturally subsystems have internal structure but their interfaces are handled on a subsystem level. Consequently, subsystem interfaces grow very wide. Wide interfaces and dependability between the cellular subsystem and the application subsystem leads to high coupling between them. The device subsystem does not suffer from the same problem because it is just a composition of rather independent hardware drivers, each having its own interface.

The subsystems are not equal. Hardware drivers and cellular software have a clear role. The application subsystem is "everything else". It becomes the controller of the phone, knowing its global state. This creates state coupling and even content coupling between the subsystems. Finally, phones are not as homogenous as assumed by the architecture. Some have special functionality (e.g., data communication) to be accounted for.

5. EVOLUTION

Initially the variance in the subsystems was mainly functional. Some phones had special features and accordingly there was a special part in the software handling it. Then the coupling between application subsystems and others started to play its role and increasingly variance in one place in the software was just a reflection of variance in another part.

Required configurations were implemented by using a configuration management system together with more fine grained mechanisms like source-code pre-processing using macros and compiler flags, or programming language based mechanisms like indirection and late binding of functions, variables and types.

The elements of variability supported by these mechanisms (text lines, functions, variables, and files) are not the elements of variability required by products (features, platform differences, interface styles). In particular, source-code preprocessing using compiler flags is problematic. It is the most versatile variance mechanism, allowing the possibility of making every source line a special case, but it does not build any abstractions. From the source, it is practically impossible to determine what each flag means, or what combinations of flags are permissible.

As the variance kept growing, it became hard to control the mapping between desired product variance and its implementation. This happened in the golden years of artificial intelligence and the "natural" solution was to automate this mapping by developing an expert system setting the compiler flags based on a list of features that the phone was supposed to have.

At the same time, the development organization was growing and subcultures started to emerge. Since each site was mainly responsible of development for a particular market area sites did not have to deal with variance in network standards. They started to maintain their own versions of application subsystems to get rid of the variation caused by multiple network standards. This confined the reuse benefits into small groups of products but it did also cut the cost of reuse and increased independence of each site.

The initial architecture proved to be very stable. A number of small subsystems were added as the mobile phones got new functions but the initial architecture maintained its central role over several phone generations.

As the phones kept evolving, new functionality was added. Now that the application subsystem was diversified, it became apparent that the cost of porting new functionality across the product family is substantial. Maintenance problems with the application subsystems also made it clear that the application subsystem had to be redesigned to be more flexible.

The redesign was carried out according to an object oriented user interface design style separating control, presentation and functionality (à la MVC). The redesigned application subsystem replaced the old application subsystems and an attempt was made to keep its interface as backward compatible as practical. This attempt succeeded. The development was carried out parallel to the product development based on the old application subsystem.

When the new application subsystem was taken into use, the organization had to be restructured accordingly. We identified three different development categories:

1. The infrastructure development group improves the application framework and ports it on different hardware platforms.
2. The component group develops reusable presentation and application components.
3. The product development projects compose their application subsystems using existing components and develop new components when necessary.

The whole process is driven by the product development projects. They place requirements on the infrastructure and request new components.

Up to this point, we had been able to accommodate each change either by adding new modules to the architecture or by reworking existing elements. Recently we had to face a bigger challenge.

Markets continued to develop and new product ideas were put into practice. This led to erosion of the basic premises underlying the initial software architecture. We could no longer assume that there would be only one cellular system in each phone since phones should be able to operate in different networks. Auxiliary equipment continued to become more intelligent. In addition to the user interface, phones could be controlled by infrared and serial connections. In one setup, the phone had to act as a central controller for a large distributed system and in another it was completely subordinate. The success of the Java programming language also pushed downloadable software to phones. Clearly, we needed a new architecture.

The basic idea behind this new architecture is to separate the service identity from the identity of its provider and make service usage and provision location independent (see *Figure 1*). With dynamic configuration management, we can have several providers for the same service and these providers can be plugged in or taken out without restarting the system. Architecture supports both local and remote message passing and object management, task scheduling and event control. This architecture is also much better described. It defines software components, message interfaces between components, essential use cases, component grouping and deployment structure. The initial architecture had only interface and runtime architecture descriptions.

The biggest challenge in this new architecture was not how to design it but how to adapt to it without stopping the product line. We approached the problem by moving into the new architecture gradually. A roadmap of new architecture versions was outlined. Each version has more capabilities than earlier ones. Architecture versions are developed concurrently with product development projects and each project is based on a version that satisfies its needs. Currently our new phones are based on the second version; a third version is being implemented, and fourth is under design.

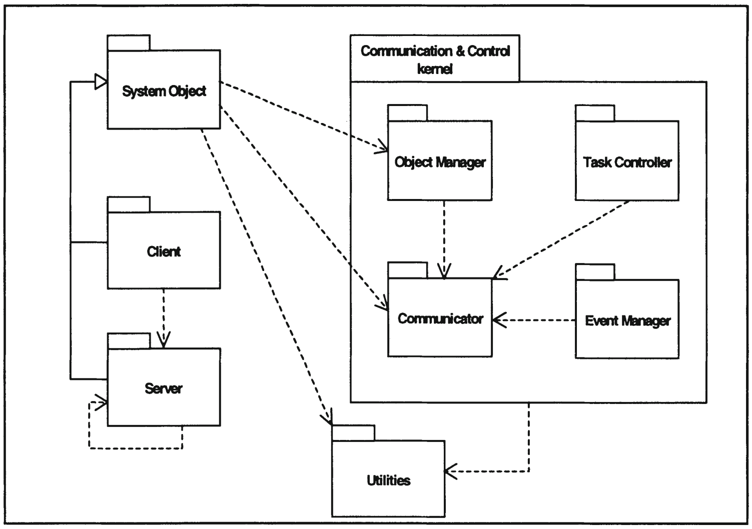


Figure 1. Basic module classes for the new architecture

This architecture evolution roadmap that was planned to help us to control the move from the old architecture to the new one is going to be permanent. It gives a view of the future and helps the product development projects to assess what will be possible and when. It is also a basis for reasoning about how to develop further the product family.

6. LESSONS LEARNED

It is often assumed that the development and management of an architecture addressing all the common requirements of a product family and providing the basis for wide scale reuse would always be economical. This is not quite true. When independent products continue their evolution, they face new requirements. These requirements can be tackled only in the product development project that control the resources and have the responsibility. Later, some of the new requirements may prove to have wider scope and they can be tackled on the family level. However, changes on the family level may be very costly for the product projects. Commonality management also requires communication and cooperation. Such a cooperation between different organization over wide distances is complex and costly. Reuse and modifiability must be balanced according to the product development organization and market needs.

Our experience shows that architectural evolution is possible and practical. We made three different types of changes.

1. Adding components, which turned out to be rather easy as long as they required no special services.
2. Redesigning components, which was difficult whether you changed the interface or not.
3. Redesigning the architecture with new communication mechanisms, a new execution architecture, and new component roles, which was very difficult and costly.

Note that all the changes were incremental; nothing was ever built from scratch.

This experience shows that architectural change has to be carefully planned, taking into account its organizational aspects. Then it has to be carried out so that product line does not stop. Large architectural changes are high-risk operations. Even when they succeed, they tend to take much longer time than expected. New products cannot wait for the new architecture.

This history also demonstrates that variation management and reuse are tightly connected. If your variation management runs into trouble, you may ease the situation by decreasing reuse. The gray area between a perfectly organized product line and completely independent development projects is wide.

It takes time to react to the changes in the domain of requirements and a practical product line is never optimal. Make a roadmap showing what you intend to implement and when. Base the changes on the needs of product projects and product concept developers. It is hard to give reliable economical justification for each change but it is easier to compare different change requests.

ACKNOWLEDGEMENTS

I wish to acknowledge the help provided by my colleagues Alessandro Maccari, Anssi Karhinen and Alexander Ran and anonymous reviewers. Their comments improved this report substantially.

Building Systems from Parts in the Real World

Everything's Possible, But Nothing is Easy

Roy R Weil
Michael Baker Corporation, 4301 Dutch Ridge Road, Beaver, PA 15009
WeilR@acm.org

Key words: Component reuse, software integration, component packaging, configuration management

Abstract: As manager of the software support group for a full-service civil engineering firm, my major challenge is in composing project-specific software solutions from pre-existing parts. Too often, the available parts are not quite right for the task; or the parts work individually but make incompatible assumptions about interaction, representation, or other aspects of integration; or a solution works on prototypes but does not scale up for production. For seven example cases, I describe the architectural integration problems and what we did about them. None of the specific examples presents a major challenge – my real problem is that each such example must currently be solved as a special case. The architecture research community could help me most by developing general methods and tools to help me identify and resolve these integration problems systematically and routinely.

1. INTRODUCTION

Baker Engineering is a full service civil engineering firm. We design and build structures such as highways, bridges, buildings, airports, and subdivisions. We have a division that operates and maintains clients' facilities. We also have a Geographic Information Systems division that creates geographic databases for clients. Some of our services are software intensive. We use engineering design analysis tools, geographic information systems, and Intranets both in-house for our engineering tasks and as products that we deliver to clients.

The software is bought, provided by clients or government agencies, and developed internally to various degrees. We often adapt software and packages from previous projects. We prefer to create projects from existing components whenever possible, resorting to new code development only as necessary. The various departmental groups within Baker typically develop these projects with consultation from our small software group. When we know in advance that the software aspect of the project is problematical and/or requires extensive internal development, our software group may do the primary software development and then turn over the project to a departmental group. These groups typically do not have a person who is dedicated to software development. They typically do have an engineer or technician who, as a sideline to his production responsibilities, provides immediate computer support to his department.

I manage the software development staff that provides software consulting services to the various other units of Baker. We usually take on projects requiring one to three man-months of effort, although we may have one or two longer-range projects going on at any one time.

My biggest ongoing problem is the difficulty of adapting existing software to new projects, creating the glue that makes separate software components work together, and scaling/hardening prototypes for production use. For example,

– Projects often create or acquire interactive tools to prototype the production phase. When the project goes into production, we must process large batches of data – an operation best done in batch. But the interactive tools sometimes don't provide enough program hooks to run in batch.
– Projects adopt tools that provide good immediate functionality but have proprietary representations. As time passes, we need to add functionality, but available components that perform the new functions can't handle the proprietary representation.
– Professional engineers (i.e. non-software types like civil, mechanical) who are not programmers may develop or adapt applications that are good for personal use and later the company wants to expand the use to other individuals or larger projects. This may involve simply adding a better user interface and more robust error detection and handling, or it may require the replacement of some components with more general or scalable ones, or it may involve re-implementing the concepts in some language/application that will scale to a larger volume or more users.
– Cost considerations sometimes force us to assemble solutions from parts or even write fresh code, even when technically adequate solutions already exists. Often the critical cost is not the initial cost of the

application, but the cost of deploying its viewers, browsers or other client components to all potential users.

In the body of this paper I describe several specific examples. Each of these can be solved in isolation; the solution is often straightforward. Indeed, the solution is often included in the example. The major challenge I face as the manager of the software development group is not solving any particular problem, but the sheer volume of problems like these and the drain it places on our software development resources.

The important point behind these examples – the pervasive architectural issue – is that these problems come along as regularly as rush-hour traffic, and each one consumes resources, sometimes significant resources, for its special-case solution. Further, we generally spend more resources in discovering the incompatibilities and shortfalls, than in fixing them. The consequence is that we don't realize the promised benefits of component reuse, and we get less done at more cost than we should. Even worse, there are projects we would like to do but do not even try, because we can see how the petty aggravating integration details will more than offset the benefits.

2. EXAMPLES

2.1 LIDAR: Improved aerial mapping

2.1.1 Problem background

Lidar is an aerial mapping technology for producing high precision contour maps to be used in engineering design. A specially equipped airplane uses prototype technology for high-precision data acquisition. Software developed by an external research organization does the initial conversion from data acquisition to xyz coordinates. We (Baker) had to take this software and turn it into a production product that would efficiently process over 30 Gigabytes of data per county scanned. Plans are in place to process over 50 counties in the next year.

The airplane acquires data in the form of scan lines at 3-meter spacing. Points along the scan line are 3-meters apart. This gives us a grid of points 3-meters by 3-metters. Each point consisting of the distance from the airplane to the ground, the GPS (Global Positioning Satellite) location of the aircraft, and the roll/pitch/yaw information for the aircraft. The data must be converted to topographic maps by the process:

- Do internal calibration and correction of the data acquisition system
- Calculate the <x, y, z> location of each spot on the ground through spherical trigonometry

- Transform the <x, y, z> location for geographic projection
- Produce a contour map
- For quality control, produce difference maps for overlapping areas; also compare the contour maps to known low-precision maps
- Merge adjacent flights where the data overlaps

We use a mixture of pre-built and custom software to accomplish this.

2.1.2 Computing obstacles and resolutions

Contouring package format incompatibility. The program was originally developed on a Sun workstation. The developer created <x, y, z> output ASCII files that were formatted for a graphical display package. The display language provides commands to specify scale, viewpoint and sun angle. It allowed comments with a # sign in column 1. The contouring package wanted the data in <y, x, z> order and comments with a semicolon in column 1. The initial workaround was a Perl script that did the format conversion. A better one was a parameter to the conversion program that told it which format to output. This of course required us to rework the program to find all the places that output data.

Difference package worked in interactive mode, but failed in batch. Some of the flights covered the same ground area. In particular one of the early flight was flown perpendicular to most of the other flights. By checking this flight against later flights we could verify that the data acquisition system did not drift over time. We do this check by calculating the difference between the two generated ground surfaces. The difference program worked fine in the interactive environment, but produced no output when I tried to run it in batch. Since I was planning to do about 8,000 of these differences, this was unacceptable. I sent data to the vendor and they were able to reproduce the problem. It took three weeks to get the fix back. They sent me the patch, but it was for the wrong package. The software we use is structured in 12 to 15 sub-packages each with specific functionality. You buy as few or as many of the sub-packages as you need. There is overlap in the functionality. The difference routine was in three of the sub-packages. They sent me the patch for the only one of the three that I had not bought. The workaround was to get an evaluation serial number for the unpurchased package. Hopefully they will issue the update to the package that I have purchased before the evaluation period runs out.

Display package couldn't handle volume. Because of the high density of the data, (a point every 3 meters) the contouring package produces lines that are very ragged, rather than the smooth lines that are on a typical contour map. This makes the display files 20-50 times larger then usual, and they exceed the allowable file size of the display package. The workaround was to process the data in sections. Two programs were written: one to split the data into reasonable-sized chunks, and the second to tell the display package to display the multiple files required to cover the specified area.

Coordination with our partner company. The development was a joint venture with an external research organization. Each group had separate responsibilities. We had a difference of interpretation with the partner company. The original data met the specifications but showed a systematic error. In our opinion, the specification was not quite right, because it allowed systematic, correctable errors. The specs called for an error of plus or minus one foot. The data showed a systematic error of plus 6 inches on every tenth scan line. There was a physical explanation for the error. One of the pieces of the apparatus was a spinning ten-faceted mirror. Since it was the partner company's responsibility to produce the <x, y, z> data, I thought it was their job to determine the exact magnitude of the error, and produce the 10 adjustment factors. Their response was that the data was within specs and therefore they were not going to do anything about it. Since I knew the eventual clients of the data would not accept data with this kind of error, I had to modify the partner's program to accept the adjustment factors and to determine the values.

Dropping information too soon in the process. To make the problem of removing the systematic error a little more interesting the data acquisition hardware reported out the rotation angle of the ten-faceted mirror in a 17-bit integer. Since the original programmer did not care which facet was being processed, he simple extracted the low order sixteen bits (2 bytes) from the data stream to obtain the angle of the mirror for each data point. This meant that when I needed to determine which facet was being processed, I had to determine where in the data stream the 17 bit was located and merge it with the other 16 bits before determining the facet number. Obviously a better interface between the modules would have made this task simpler.

Scaling up the data sizes. When the program was in the developmental stage the length of a data collection stream was relatively small, less than 2 miles of data. At a sample rate of 30 points per second, this produced 1300 points and the programmer declared his arrays and structures at 2,000. The production environment uses longer flight lines. There turns out to be a system constraint of about 35 miles. These limits were nicely “`#defined`” in C so I upped the limit from 2,000 to 32,000 and got “program exceptions” for some medium size cases. It turns out when you do a `malloc` of a double precision variable (8 bytes) for 2,000, the answer (16,000) is less than 16,384 so storing the `malloc` calculation in a small integer works. I had to find all the `malloc`s (there turned out to be 23 of them) and change the calculated variable from an integer to a `long`. And, of course, after I had set the limit to 32,000, 5 flight lines out of about 400 exceeded that limit. This required increasing the limit above 32,768 to process those lines. I had to revise the program so that all array accesses were done with long integers.

Miscommunication of operator procedures. To process a flight line of data requires 7 related files. To provide for the batch processing and to reduce the chance for operator error, I devised a file-naming scheme in which the first 5 characters of the file name established a relationship. This scheme apparently was not conveyed correctly to the person producing the files. They did not observe the naming rules. This resulted in a group of files with the wrong names, which therefore could not be processed automatically. Since the data was burned onto a write once CD after being created it was not a correctable by a one time renaming of the files. To resolve the problem I wrote a rename script that the operators must execute each time they copy files from the CDs

2.2 GeoMedia Web

2.2.1 Problem background

One of our long-standing utility clients was interested in a web server for displaying geographic data. He wanted to ask questions like "where are all my open work orders" and see a display on a map with all the selected buildings filled in red. He also wanted to click a telephone line on a map and have the system trace it back to the central office. The drawing files were in Intergraph CAD (Computer Aided Design) file format. Intergraph has a product called GeoMedia Web, which will take a one of their CAD files, or a portion there of, and reformat it to a standard web format (.CGM) for which there exist plug-ins and ActiveX controls.

Another of our clients, with our help, had populated such a database and had developed the queries and Perl scripts to do those kinds of queries. They were willing to allow us to use a small sample of the data from the existing web site and demonstrate it to the other client. So the task was to take a working web site, move it to another server, and reduce the size of the data set.

2.2.2 Computing obstacles and resolutions

This task turned out to be an adventure in incompatible assumptions and hidden configuration files with hardwired information.

We installed GeoMedia Web on the new server. We copied the existing web directories from the working server and attempted to access the home page on the new server. We got a "404 page not found" error message.

Different web servers assume different default names when you request the home page. In one case it was index.htm, in the other it was default.htm

We next discovered that the GeoMedia Web program had some internal configuration files that it used to let itself know what "projects" were available. It kept these files within its own directory structure, rather than in the Web's directory structure. We had to go back to the original web and obtain this file, and then modify it to contain only the demo project, not the other five projects that were on the original site.

Another of the configuration files contained the name of the ODBC entry that pointed to the database. On the new machine the ODBC name was different and the original user id and password did not exist.

Still another configuration file contained a list of file names within the project. These names were fully qualified and therefore had the wrong drive letter and upper-level directories for the new web servers file location.

Once the cause of each problem and the associated configuration file was identified, it was straightforward and easy to fix. However, to discover the cause of the problem took some time. In general, the error messages just indicated that something did not work or was missing. It took some sleuthing, debug tracing and debug prints to find out exactly what was wrong. Finding and resolving these problems could have been easier if there had been a configuration description and updating tools.

2.3 GIS version control

2.3.1 Problem background

Intergraph offers a suite of programs for working with GIS (Geographic Information System) data. It includes a basic nucleus package, an administrator package, a digitizing package, several kinds of analysis packages, etc. In all, there are about 12 to 15 specialized packages, each of which does a collection of related tasks. The specialized packages are sold separately. These packages use a database to store some of the configuration and attribute information.

One of the things Intergraph did right some years ago was to develop an interface between the packages and the various database engines. They created an internal definition of SQL, which all the packages use. This interface package processes the internal SQL and transforms it into acceptable SQL for the given database engines that is attached at your site. Microsoft users will recognize the ODBC model. Intergraph calls it RIS (Relational Interface System).

2.3.2 Computing obstacle and resolution

I recently needed to use another one of these packages. I downloaded and installed the new package. When I ran it, I immediately got an invalid database message. Since the database in question was working with the other packages I knew that was not the problem.

Upon investigation I discovered that Intergraph was in the process of upgrading the packages to use Microsoft's ODBC database interface rather than RIS. The working packages were the updated versions, which I had associated with a Microsoft Access database using the ODBC interface. The non-working package had not been updated and would only work through the RIS interface.

To accomplish my project I needed to be able to run the different GIS packages that were at different version levels. The solution was to transform my Microsoft Access database to an Oracle database and attach the Oracle database that would work in both version levels.

2.4 Upgrade version compatibility problem

2.4.1 Problem background

I was working with a commercial package that had a number of command-line processing commands, each of which had 6 to 10 parameters. These were expressed in the Unix command line style of minus sign, followed by a single letter identifying the parameter, followed by a space, and then the parameter value:

```
task -I input file -O output file -L level
```

We had put together an extensive series of scripts to process some data. This collection contained over 10,000 lines of script. We have been using these scripts in a production environment for over 12 years now.

2.4.2 Computing obstacle and resolution

As you can imagine, over the 12 year life of the system the underlying commercial package has gone thorough a five major revisions and many minor revisions. Each revision has brought changes in the parameters and parameter letters. The changes generally made the letters more consistent amongst the various commands, but sometimes additional functionality added more parameters. To compound matters, the scripts were being run in several locations, on many different machines, which made it infeasible to do a simultaneous switch among versions.

At first, every time a new version appeared, we would install it on a development station and proceed to test the scripts. When an incompatibility was found, tests were inserted into the scripts to determine which revision was running. Based on the revision number different sections of the script were executed. After a while the scripts got pretty ugly looking, and it became hard for maintainers to follow.

After the third major upgrade the software/scripts were stable enough that we did not install the next two upgrades. So we are now running on packages that are two versions (about 6 years) back. This gives us grief when problems do occur because we can not get any support from the vendor. In addition the hardware maintenance cost on the older machines is expensive compared to the purchase cost of today's machines. At some point the maintenance costs are going to force us to redevelop all that software.

The overall problem is that the vendor's architecture did not provide a migration path between versions. Nor did it provide a reasonable way to run multiple versions of the same software on multiple machines with shared scripts.

2.5 Image library

2.5.1 Problem background

The Corporate Communications Department wanted to make available a collection of public relation images for projects to use. These were photographs of past projects, generally of high quality. These photos would be used in marketing brochures and in proposals to other clients.

One of the primarily software selection criteria was the mandate to allow as many people as possible within the corporation to have viewing, selection and download capability of these images.

We considered several packages. Most give us the required functionality of filing an image, assigning keywords to that image, performing a search on the keywords, displaying thumbnails of the search results, and downloading selected images. Of those that met the requirements the costs ranged about $2-5K for a server and $20-30 per client plus the cost of deployment. For the corporate communications application we estimated a need for 100-200 clients. Other departments could also use a similar system for the storage and deliver of drawing details, inspection report photos and progress report photos.

Instead of purchasing one of these systems, we decided to implement a package in-house using a web-based approach. This was almost purely a cost decision, since some of the packages did exactly what we wanted. The difficulty of maintaining client viewers on several hundred workstations was also a deterrent.

2.5.2 Computing obstacle and resolution

The in-house application was built from scratch. It uses Microsoft's active server page technology to access a database that contains information about the images. The images are stored in a directory on the web server. The system consists of about eight or nine different web pages that are populated on demand from the database.

One of the web pages consists of a display of a number of images that had been selected by a search request. To make the system work at acceptable speeds this display had to show a thumbnail view of the images. The thumbnail would be a 96 by 96 pixel image that would download and display very quickly. By clicking on this image the user could obtain a larger, more detail picture.

We found three free or cheap applications that would convert a directory of high resolution images to a set of thumbnails. However, all were interactive. An operator had to select/display the images, invoke a command from a pull-down menu and perhaps enter new file names and/or directory. One of these applications would do some commands via a DDE interface. Given a DDE interface it is possible to write code that will call the application in a batch environment. However, the application did not expose the functionality for the process that we were executing.

Since we were dealing initially with only 500 to 600 images, we decided to grunt it out. We used the macro facility of PhotoShop to do the re-sizing and naming work. The operator loads the image, invokes the macro, and then goes on to the next image.

As a side note: After the 600 conversions to thumbnails were completed, another application became available that allowed the creation of the thumbnails in a batch environment.

2.6 Resume library

2.6.1 Problem background

The Document Services Department maintains a collection of resumes of the professional staff to be used in writing proposals. The resumes are formatted electronically in WordPerfect, using a house style, including logos. They are available on a shared server. Proposal-writers who want to incorporate a resume into proposals could attach to the server, copy the relevant WordPerfect file to their hard drive and make the necessary changes. In general, these changes would be textual, not format. The changes might include deleting paragraphs that were not relevant to the proposal and adding paragraphs to emphasize relevant experience. Once the proposal is submitted the modified version is discarded.

The task was to make this collection of documents more widely available, easier to search and easier to obtain. The obvious choose was to make the data available on the internal Intranet.

2.6.2 Computing obstacle and resolution

No available WordPerfect-to-HTML converter works well enough. Everything we could find butchered at least the house formatting and sometimes everything else as well.

We needed a full-text search. Sometimes a proposal author wanted to include a resume of an individual with a particular expertise that was not commonly exploited by the company. In such a case knowledge of who had that experience would not be available from mental memory.

Although the resumes are loaded on a central server, the updating is done throughout the company by the various departments and individuals.

To avoid the problems with multiple copies of the same information, the resumes were kept on a single shared server. Typically the remote offices would not be attached to this server, so they would have to go through the procedures to attach to the server before they could access the data.

We examined several alternatives:

There exist web browser plug-ins (KeyView, QuickView) that have the capability to display the Word Perfect format. However at $30-50/client plus installation, the plug-in route is expensive, both in initial outlay and in distribution/installation/update costs.

PDF format is another alternative. The user would view the PDF formatted document, but download the Word Perfect version for the proposal. Version control is the main reason that this was rejected. Updating is done rather frequently and by many people in the WordPerfect formats. Getting the updater to create the PDF file whenever a change is made is problematical, and would require the Adobe Distiller at many desktops. Automatically invoking the PDF converter would have to be programmed.

Another package (Net-it Central) converts any printable document into its own proprietary format (.jdoc), which is displayable by a browser using a Java applet. This package will work in a scheduled batch mode, converting any documents that have changed since the last time the package was run. This batch process creates a table of contents for the documents. However, the resulting table of contents isn't wholly satisfactory. Net-it Central does provide a template facility to modify the header/footer/table of contents of the resultant pages that it is creating. This solution adopts yet another document format, and yet another user interface for controlling a display. The buttons for moving around the document are separate controls, not integrated into the browser buttons or scroll bar. It is yet another interface for the user to know and use.

Any of the above solutions will work. None of them, however, would give us an increase in functionality or availability over using the shared server great enough to warrant the expense and aggravation of the change. As a result, we decided not change existing procedures.

2.7 Database communication

2.7.1 Problem background

A database server provides verification of customer information for small retail companies. The current system has one server and N clients. The user works off-line preparing query data. The client machine then calls the server via modem. Once a connection is established, it sends the query, retrieves a report, and hangs up. The user then displays/prints the report off-line.

The system has been in place for a number of years and was designed and built before the widespread availability of the Internet. The users are becoming more sophisticated and they want to replace the telephone connections with an Internet connection.

2.7.2 Computing obstacle and resolution

The best of all possible worlds would be to find an existing Internet interface package that provides the same procedure calls as the modem interface package. At present we have not found such a package.

One alternative is to re-write the modem interface, to establish the connection using sockets. The calling sequence and actions are roughly the same whether you are using a modem or a socket. The sub tasks are to link up with the remote machine, login a session, pass data back and forth, and then disconnect.

Another alternative is to strip out all the modem interface code and replace it with remote procedure calls directly to the various routines in the server module that do the work, then use the Microsoft DCOM concept to establish the connection between the server and the client.

This task is on the low priority list so we are still looking at technologies and evaluating our alternatives.

3. DISCUSSION

As mentioned in the introduction my biggest ongoing problem is the difficulty of adapting existing software to new projects, creating the glue that makes separate systems work together, and scaling/hardening prototypes for production use.

The examples show some of the problems that I deal with on a day to day basis. They illustrate the classes of architectural problem that make a large fraction of my work:

- Interactive programs and procedures are sometimes difficult to convert over into a batch environment.
- Different components sometimes can not be joined together even though the data that one component is outputting is of the type expected by other. The problems are both simple, like format and sequence of information and more complicated, like incompatible demands and assumptions.
- Reasonable existing solutions may be precluded by economic restrictions. Either projects just do not get done, or in-house solutions are developed which have a limited but adequate set of capabilities.
- Vendor upgrades will generally cause problems for existing systems that are built on top of the product. Vendors tend not to have backward compatibility.
- Software may not scale in size or performance.
- Not all default assumptions are documented, or their documentation is scattered in none obvious places.

I am trapped in a Turing tarpit: Everything is possible, but nothing is easy. The tarpit is not one of creating individual programs to use as components: that's pretty straightforward now. The tarpit is filled with the glue that we create ad hoc to stick together components. The glue is necessary either because the parts were not initially designed to fit together (different vendors), or the connection tools are primitive (command scripts), or the style of use does not meet production needs (interactive vs batch).

ACKNOWLEDGEMENTS

The work reported here was, for the most part, in-house software development and configuration for Michael Baker Corporation and was supported by the company. I wish to thank Joel Platt for one of the examples. I wish to thank Mary Shaw for her encouragement in getting me to write this paper and for making red marks all over the pages on many drafts.

TECHNIQUES AND METHODS FOR SOFTWARE ARCHITECTURE

Architectural Concerns in Automating Code Generation

L. F. Andrade, J. C. Gouveia, P. J. Xardoné and J. A. Câmara
OBLOG Software S.A.
Alameda António Sérgio 7 – 1 A, 2795 Linda-a-Velha, Portugal
{landrade, jgouveia, pxardone, jcamara}@oblog.pt,
tel:+351-1-4146930, fax:+351-1-4144125

Key words: Code generation, object-oriented modelling, contract-based architectural style

Abstract: We report on the problems (and solutions) that we have been facing in defining an architecture that enables us to automatically synthesise production code (COBOL, CICS, SQL) from a higher level specification language that includes both primitives that handle business and architectural requirements. Our experience has been drawn from a real-life project in the banking industry where object-oriented models for large-scale projects were used. With these models, the application architecture was conceived to be robust to change, accommodating new behaviour in a systematic and encapsulated way.

1. INTRODUCTION

Critical aspects of today's banking management information systems include time to market (dealing with component development and re-use), evolution (volatility of business requirements), requirement conformance (take decisions upon correct information), scale and complexity of systems, parallelism, maintenance, robustness and security. Product distribution, management information systems and decision support systems are typical banking applications facing these problems.

A particularly acute aspect of the problems that financial companies are facing today is the need for a technology migration plan from current traditional systems to future open systems. An encapsulation mechanism to hide "legacy systems" is essential to guarantee a smooth transition, coping with the business support extensibility.

Our purpose in this paper is to report on the experience that we have had in the combined use of formal architecture and transformations for assisting the migration of a banking application. More specifically, we will discuss the role of architectures in enabling us to automatically synthesise production code (COBOL, CICS, SQL) from a higher level specification language that includes both primitives that handle business and architectural requirements.

Our approach is based on the use of object-oriented models for large-scale projects. By using such models, the application architecture can be conceived to be robust to change, accommodating new behaviour in a systematic and encapsulated way.

In section 2, we briefly introduce some of the requirements and describe the banking project itself. In section 3, the main problems that we had to face are identified and the possible solutions are discussed. Finally, in section 4, a technique for automating production code based on transformations applied over a chosen architecture is presented.

2. PROBLEM DESCRIPTION

The global purpose of the project at hand was to migrate and improve the information system of a European mid-size bank with the following characteristics:

- 430 branches and 5000 PCs;
- 1 million transactions per day (average);
- 2 seconds of maximum response time;
- System hardware - IBM Mainframe;
- System software - MVS, CICS, DB2;
- Language – COBOL, SQL.

Our task was to migrate and improve the Retail Network, which meant re-construction of the Branch Transaction System (more or less 90 transaction types – opening accounts, withdrawals, deposits, transfer orders, etc.).

The main business requirements for this project were to:

- Improve the system functionality to deal with the new European currency (EURO);
- "Solve" the year 2000 problem;
- Adapt the system in order to inter-operate with a new package that manages the "financial products" offered by the bank;

- Adapt the system so that the bank could be open 24 hours a day (mainly because of Internet access).

The main implementation requirements for the project were:

- The client tier could not be changed, meaning that the format of all communication messages (between the client and the server) had to be preserved;
- The target technology (MVS, CICS, DB2, and COBOL) was fixed;
- Some functionalities, like check-digit validation, time-stamps, etc. were supplied by already existing routines which we were obliged to use;
- The format of communication with other modules was fixed and not changeable;
- All the technical documentation formats were also fixed and had to be followed;
- Some customer implementation techniques had also to be followed.

The kinds of problems we had to face in this project are very common to real projects. Even with the availability of many commercial CASE Tools supporting object-oriented methods (and in particular supporting UML), our main problems were to come up with answers to the following two questions.

1. How to synthesize the final production code automatically from the high level specifications?
2. To achieve the previous goal, what language/method should we follow to specify the system, including all of its details?

3. OUTLINE OF THE MODELLING APPROACH

An obvious answer to the second question above was to choose UML because it is a standard visual-modelling notation that is already in place. However, from our experience, in order to use UML it is necessary to have confidence in all of the notations and techniques that are offered. Furthermore, integrating such techniques and notations seems to be a difficult task, the feasibility of which needs to be demonstrated particularly if the goal is to automatically obtain code from specifications.

Given these caveats, we decided instead to use a rich, yet integrated and precise subset of the UML notations – which we called OBLOG – adding to this subset a rigorous and formal specification language supporting the generally accepted OO key properties of

- support for encapsulation of services and state as objects
- the ability to create object instances from class templates

- the ability to define new object templates by monotonic modification of existing ones (base classes)

OBLOG introduces new specific features such as

- integration of the concept of module in the class concept as a way to introduce different levels of abstraction and encapsulation;
- specialised language constructors to define object behaviour at distinct levels of detail;
- visibility of objects defined by contracts as an architectural style for the construction of complex systems.

These features are supported with full integration of graphical diagrams and textual specifications.

- The graphical notation is compliant with the UML standard.
- It allows for a continuous path from high to low-level design specifications, always using the same specification language.
- The textual language is mainly used for the design details.

An effort was made to provide OBLOG with a well defined semantics as a means of supporting key aspects of object-oriented construction such as method composition and extension, direct object interaction and event multicast, behaviour inheritance, composability, and encapsulation. Some properties relevant to wider software engineering were also included, namely the ability to specify concurrent behaviour properties and to deal with non-normative behaviour (exception handling); allowing the systematic refinement of specifications to code, preferably in a compositional manner.

According to these principles, an information system is treated as a collection of interacting concurrent objects. An **object** is an abstraction of an entity with a persistent *identity*, a *public interface* defined by the provided services and recognised events and an *internal body*. The internal body includes hidden *local methods* implementing the public interface, possibly calling some hidden local auxiliary services, a *computation state* indicating the object situation in its life cycle, an *internal state* (represented by its slot values) storing the effects of method executions, *hidden enabling conditions* constraining services and reactions, and *hidden invariant conditions* constraining state changes.

In practice, specifications tend to involve a large amount of objects which makes understanding and managing them a real problem. OBLOG deals with this problem by providing a decomposition mechanism that allows complex objects to be defined that can be later detailed in terms of other simpler objects.

The ability to decompose specifications also allows the analyst to introduce new objects at any level of the specification. The way of making those new objects, introduced locally for a given complex object, visible to other objects, is through a **contract** mechanism. In this sense, contracts are used to enrich the interface of a certain object, allowing some of its components to be seen by others.

In the following example (Figure 1), an *Account* object makes a contract with a *Customer* object (named *CtWithAccount*). This allows for public objects defined in the decomposition of Customer (e.g., *CustomerProfile*) to be used by *Account* and any object in its decomposition. On the other hand, the contract *CtWithCustomer* allows for obligations to be defined between Customer and Account.

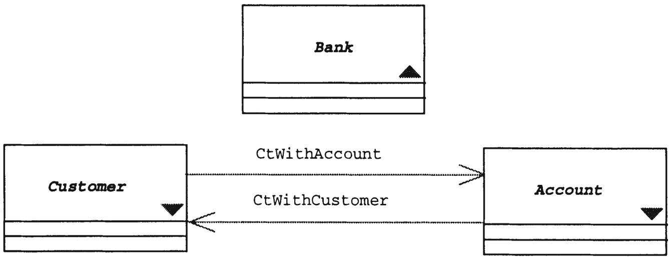


Figure 1. Contract between Account and Customer

Contracts are the privileged architectural style for OBLOG specifications. They are used to express *dependencies* that characterise *collaboration* relationships between objects. These kinds of relationships allow the analyst to perceive the way objects work together in order to perform some task.

Table 1. Withdrawal obligation requirements

Detailing obligations to define the *withdrawal* requirements		
Involved concepts	Customer obligations	Account obligations
X : Account Y : Customer Z : Amount W : Balance	Customer Y owns Account X Balance of Account X is greater than Amount Z	Balance W of Account X is decreased by Amount Z
	owns (Y, X) X.balance >= Z	X.balance = X.balance - Z
Deriving formal specifications from the above requirements (Account *withdrawal* operation declaration)		
PRE-CONDITION: ?owns(self,Y)=TRUE AND self.Balance() > Z		
OPERATION: withdrawal(Y : Customer, Z : Amount)		
POST-CONDITION: self.Balance() = old.Balance() - Z		

Contract-based architectures are also used for evaluating the *impact of changes* and to maintain *traceability* of concepts. In fact, when objects contract between them the components they need, they are explicitly, creating strong dependencies between them. These dependencies are of the outmost importance, and constitute a very important input when analysing the impact of changes in a model. Contracts between objects enable the OBLOG tools to check for those dependencies and to make available to the analyst, at any level of the specification, detailed reports about them.

In order to achieve the production code generation, OBLOG provides some concepts to define detailed behaviour of operations and interactions.

Object interaction can be direct (calling a service operation of a known target) or indirect (event multicasting). Events are incidents (or *stimuli*) requiring some response.

An operation may be classified as

- *service* - executed by direct demand of a caller object
- *event reaction* - starts method execution for every object that recognises the event
- *self-initiative action* – internal operation initiated only by the owner object, when some condition holds

Conditions may constrain operation execution. *Enabling conditions* take into consideration the internal state of the object, avoiding invariant violations. *Preconditions* are specified only on the service and event parameters, indicating the operation client obligations when using that service, and giving no guarantee about the result of an operation if its method is executed outside them.

An operation execution is supported by a main *method* and a set of possible alternative *methods*. The main method is the one selected for execution whenever the operation happens. Only when the main method can't execute due to its enabled conditions, the object tries an alternative method for that operation, if defined.

Methods are composed of *local variables* and *quarks* that exist only within the method scope, and during a method execution. A *quark* is the minimal unit of object dynamic specification, with a guard condition and a body responsible for the effect on local state and interactions.

4. OUTLINE OF THE PROPOSED ARCHITECTURE

Having chosen a set of concepts that is rich and precise enough to build the intended models (as an answer to the second question posed at the end of section 2), the problem is then reduced to the following questions.

- What architecture should be chosen for COBOL/CICS/DB2 applications in order to support these concepts?
- How do we automatically synthesise production code from the defined architectures?

- How would we be able to easily interact with already existing applications, with that interaction clearly and rigorously expressed in our models?

  Figures 2 and 3 give an overview of the approach that we implemented.

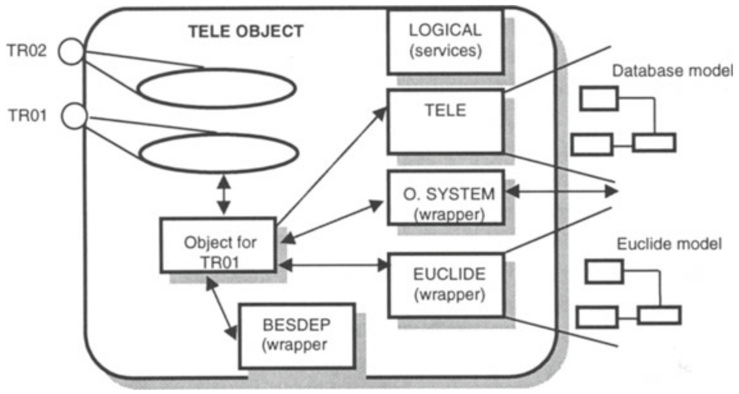


Figure 2. Logical architecture

This first figure shows the way we conceived the logical architecture for the project. The whole model is seen as an OBLOG object and therefore has an interface; in this interface we declared the set of transactions that could be called from the clients. Then, for each transaction, we created an active object that implements it, and used a delegation mechanism to direct the client calls to the right server object.

Having an object for each transaction enabled us to locate in those objects all the auxiliary operations needed for the transaction, implementing specific behaviour for that transaction. General business rules were implemented in a separate object.

We also defined two kinds of auxiliary objects.

1. objects that implemented wrappers to the data persistence mechanism (object named DB), to the external applications we had to interact with (objects EUCLIDE and BESDEP), and to the operating system (object SYSTEM)
2. an object that aggregated all of the general business rules that were used by some (or all) of our transactions (object LOGICAL)

Objects of the first kind gave us an invariant on the environment, allowing us to develop the code of the transactions without having to concern ourselves with the external changes that could have an impact on our implementation. The second object had a similar objective in the sense that it was designed to ensure that all the general business rules were fulfilled by all transactions, and that changes in those rules would have an immediate impact on all transactions.

Between the wrappers referred to above, there was one that hid the data persistence mechanism, one for which we will provide a little more detail. The OBLOG language has a set of primitives that enables the software engineer to directly manipulate the storage and retrieval of objects from disk. Though it seemed to be the natural solution to implement data persistence, there were several reasons that led us to take a different approach, implementing it as an external object.

- Data persistence is frequently a delicate point in time-critical systems, where fine tuning is often needed for performance reasons.
- In the first stages of development it was not yet decided if all of the data persistence was managed by DB2 or if we had also to deal with VSAM files.
- Using a wrapper was already the chosen solution for other "collateral" problems.

In fact, the wrapper that was hiding the data persistence ended up being developed as an OBLOG model on its own, and all of the data access code was generated automatically.

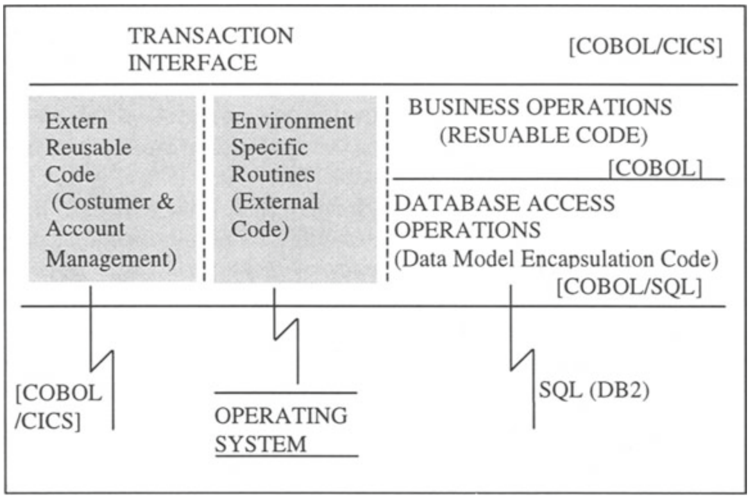

TRANSACTION INTERFACE
[COBOL/CICS]
Extern Reusable Code (Costumer & Account Management)
Environment Specific Routines (External Code)
BUSINESS OPERATIONS (RESUABLE CODE)
[COBOL]
DATABASE ACCESS OPERATIONS (Data Model Encapsulation Code)
[COBOL/SQL]
[COBOL /CICS]
OPERATING SYSTEM
SQL (DB2)

Figure 3. Physical architecture

The physical architecture is organised in terms of different layers, each of which takes care of specific tasks.

- The first layer is the interface for the clients and implements the logic of each transaction (basically, a sequential composition of logical services).

This first layer is coded in COBOL with CICS instructions for transactional purposes.

- The second layer contains all the general services that are reused by the transactions. This layer contains COBOL code only.
- The third layer implements all the data persistence operations and is coded in COBOL with SQL instructions.
- Also present are modules that implement the connections to exterior services, whether they be operating system services (e.g., get current date), standard services (e.g., check-digit validation) or accesses to other business applications (e.g., validate client information, perform changes on account balance).

In summary, the underlying ideas for the proposed architecture were

- choosing an architecture for COBOL programs/transactions in a way that they can be seen as a composition of sub-routines, each one implementing an object operation from the specification. With this strategy, we can achieve encapsulation of objects.
- Although polymorphism cannot be implemented using traditional COBOL, a certain degree of inheritance (in the perspective of code re-use) can be achieved using a code inclusion mechanism.
- Persistence of objects can be managed by a relational database (DB2). For this purpose, a model object can be created for ensuring data persistence, encapsulating all of the data accesses (either supported by DB2 or by any other mechanism). The database and the access to SQL tables are then defined through a module whose interface consists in creation, modification, retrieval, and deletion operations per object, hiding internal optimisations;
- All external components interacting with our system are isolated, in a systematic way, clearly defining the communication points and avoiding undesired collateral effects.

The next section describes how, given such an architecture, code can be automatically produced from specifications in a rigorous and continuous process.

5. SYNTHESISING PRODUCTION CODE

Automated code generation is a goal developers have been trying to include in project life cycles for a long time. It is usually viewed in two ways

1. as a feature that produces only part of the expected result, which makes it largely unused (the pessimistic view), or

2. as a feature that, when well managed, can bring significant productivity gains (the optimistic view).

The most common view is probably the first, due to the inability of current tools to address some key points in automated generation. The usual problems that real projects face are, among others:

- Insufficient code is generated (most of the times only templates).
- Code tuning may not be preserved on consecutive generations.
- Incremental generation is not available.
- There is a lack of strong customisation facilities.

We use the expression "automated generation" in the general sense of producing automatically any written information from already built models. This typically includes source code and several kinds of documentation.

With the OBLOG tools we have addressed the previous problems using the following key features.

- use of a rich specification language
- use of an open repository model
- support for customisable repository query/report technology and tools

We now discuss this technology, its principles, and how it was used in the project for enabling code generation.

The generation is a *transformation process based in rewrite rules*. A model transformation process is the application of rules to a set of objects in a certain order and according to a given strategy. The principle of a rule is to define an elementary transformation on repository objects. By transformation we mean the process of querying the repository and generating information from it, either in a textual form or into another repository (possibly the same).

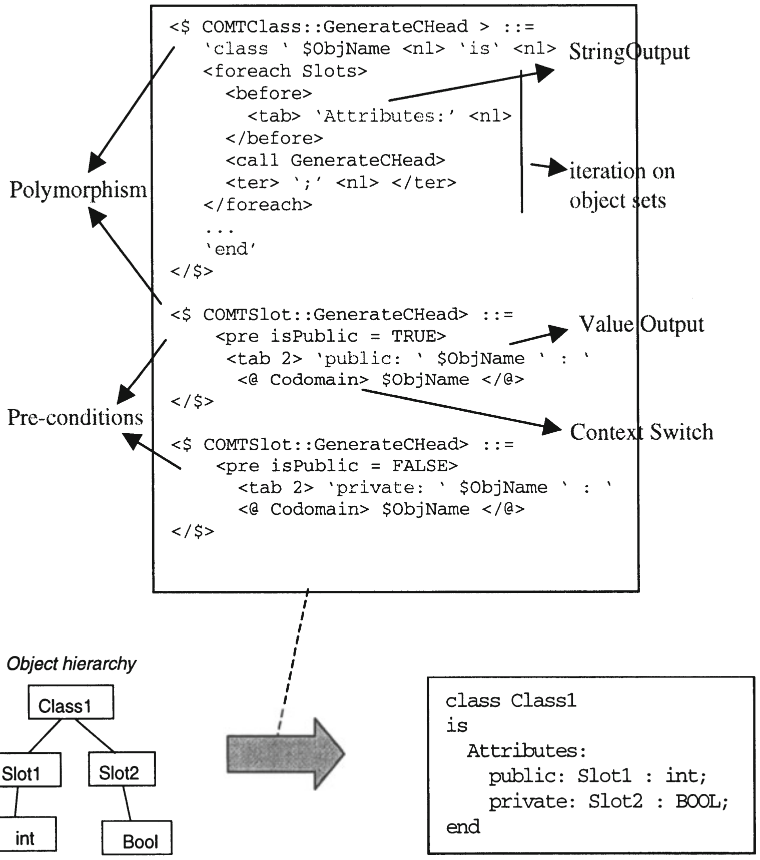
In the following we briefly present the characteristics of the rule language, with small examples.

- A rule perform actions. These actions are of several types, including elementary actions to output text and values to files, creation and modification of objects, output to user, manipulate variables, etc.
- Rules can access object properties and relations, including the repository hierarchy.
- A rule can use other rules. This allows for rules to be applied in isolation, or to be used in a call sequences (procedural way).
- Iteration mechanisms are provided to iterate over lists of objects, the repository hierarchy, numerical intervals, etc. Several groups of actions may be defined on iterations to help the processing of a list, like actions to be executed before/after the iteration, as a separator/terminator of each element, etc
- A rule always has a context object of execution. This provides an object-based view of rules. Whenever a rule is applied there is always an

underlying context object. When a rule uses another rule it is implicitly applying it on the current context object.
- Some actions may change the current context object, like when iterating through a list objects. An explicit way to change the context to a given object is also provided.
- Rules are polymorphic (on the invocation context object class). A rule body may be defined to be applicable only of objects of a certain repository class. The same rule may have several bodies for different classes. According to the context's class the corresponding rule is applied at invocation time.
- Rule bodies may have pre-conditions. At invocation time, only the rule body for which the condition is true will be applied. If no condition is true, the rule fails and a warning is given to the user.
- Rules have a simple organisation structure based on hierarchical modules

A particular kind of model transformation is source code generation and project documentation. In this particular case we used transformations from OBLOG representations to RTF and HTML in order to obtain, automatically, all the project documentation. And we used transformations from OBLOG representations to COBOL, CICS, SQL to automatically obtain source code. Figure 4 illustrates some of the above characteristics using a simple text generation. The following example concerns the rules to generate DDL code, from which we present a simplified fragment.

```
<$ public ddlCreateTable> ::=
<! "Creates one table">
  <foreach Slot>
  <before>
    '-- TABLE ' $ObjName <nl>
    'CREATE TABLE '$ObjName ' ('
    <? v_targetDB = K_DB2> ' \\ ' </?>
    <nl>
  </before>
    <call ddlTableAttributeCreation>
  <sep>
    ',' <? v_targetDB = K_DB2> ' \\ '</?>
    <nl>
  </sep>
  <after>
    <? v_targetDB = K_DB2> ' \\ ' </?>
    <nl>')'
    <? v_targetDB = K_SQLSERVER>
      <nl>'go'
      <else> ';'
      </?>
      <nl 2>
    </after>
    </foreach>
  </$>
```



```
<$ COMTClass::GenerateCHead > ::=
   'class ' $ObjName <nl> 'is' <nl>
   <foreach Slots>
     <before>
       <tab> 'Attributes:' <nl>
     </before>
     <call GenerateCHead>
     <ter> ';' <nl> </ter>
   </foreach>
   ...
   'end'
</$>

<$ COMTSlot::GenerateCHead> ::=
    <pre isPublic = TRUE>
     <tab 2> 'public: ' $ObjName ' : '
       <@ Codomain> $ObjName </@>
</$>

<$ COMTSlot::GenerateCHead> ::=
    <pre isPublic = FALSE>
       <tab 2> 'private: ' $ObjName ' : '
       <@ Codomain> $ObjName </@>
</$>
```

```
class Class1
is
  Attributes:
    public: Slot1 : int;
    private: Slot2 : BOOL;
end
```

Figure 4. Some of the rule characteristics

In a real project, the ability to customise the generated output is a key issue. It is not reasonable to think that a general code generator can serve all needs. Several aspects may contribute to this.

- A project may use a very specific underlying architecture, and specific (non-standard) target languages.
- There are proven results for some design and architecture patterns that work better in certain kinds of systems.
- Each organization has its own rules and standards that must be fulfilled.

The full customisation of code generation according to well defined architectures, enabled us to develop the project with a consistent high quality level.

OBLOG provided us with a default set of rules that performed a standard code generation for a CICS/COBOL/DB2 architecture in a MVS environment. However, those rules had to be changed to meet the project needs, in terms of naming normalisation, machine-dependent details, project dependent constraints, etc. In fact, during the development phase of this project we had two major examples of almost importance concerning the ability of easily customising source code generation.

First of all, as we said previously, we had to preserve the format of all communication messages sent by the client. In those messages, together with the semantically important parameters, there was some machine-dependent header information (totally irrelevant to the specification) and some obsolete parameters. To make things even more complicated, all of the message was compacted in a stream of characters that was the only real parameter that physically arrived at the server, and that was supposed to be sent back.

In our models, it made no sense to declare both the obsolete parameters and the machine-dependent header parameters. Moreover, we didn't want to embed in the specification some machine-dependent details. However, the idea was to generate executable code from the models, and in order to do that, those architecture-specific details had to be somewhere in our specifications.

We solved this problem by acting upon the set of generation rules, defining new rules where those project specifics were expressed. This way, we were able to design "clean" models, where only the semantically relevant information was defined and no "noise" was introduced, and yet we were able to automatically obtain the production code, with all the needed particularities.

The second major problem was a self-inflicted one. Our project was, at first, designed to implement only OLTP branch transactions. As we managed to do that before the scheduled date, our prize was to implement all of the batch transactions as well. When we made the analysis of those transactions, we realised that most of the business rules that were used for the OLTP transactions applied to the batch ones. This sounded like good news to us, but again we had an architectural problem to solve: the "call" mechanism that we used for communicating between the OLTP transactions and the object that provided all of the general business rule validations was not acceptable for a batch process, because it was too resource consuming (we had about 1500 calls in our models). As the code generation algorithm is expressed in a set of rules, it was very simple to change the call mechanism into a "code inclusion" mechanism, with no changes on the specification, coping with the environment constraints.

There were other project activities where we envisaged the use of this technology, namely model validation and impact analysis, although we did not apply it to its full extent. In terms of impact analysis, it is possible to produce reports on the interdependencies between objects, in particular the ones that are system-critical in terms of change management.

The rule engine presented above also provides a mechanism on which rules may be executed under the control of an *integration model*. This allows for transformations between a source model and a target one to be recorded in a way that they can be re-played later. In this way, consistency between source and target can be maintained much more easily. Traceability reports on transformations or inconsistency reports are easy to produce.

In the context of this project, this mechanism was applied to integrate the conceptual object-oriented OBLOG model and the relational database model. Rules were provided to transform a class model into a relational one, over which the DDL/DML generation rules were applied.

We also want to stress that the power of using specific queries on the models is increased by the facility that the tool provides in categorising objects, and relations between them, in many different ways. By classifying the objects according to some user-defined categories (e.g., architectural, persistency, interface) the application of transformation rules is much more flexible and targets correctly each object role in the system.

6. CONCLUDING REMARKS

To really have a continuous process from specification to production code it is mandatory to have

- a rigorous specification language with concepts for business as well as architectural requirements, a methodology to explain how to use these concepts to create models, and a computational tool environment
- a process to obtain production code 100% automatically generated from the specification models

- a flexible mechanism (based in interpreted re-writing rules) to query the repository meta-models to make the necessary transformations

In this approach, extendible transformation rules play an important role, because they are the only way to precisely incorporate in a generic software development process the specifics of a particular project.

Recall the three axes approach to the software development cycle (see figure 5). The solution must be clearly expressed without any concerns with the system architecture or the target software environment. Then, the solution must be matched to a given system architecture, and the target software environment must be chosen, so that the appropriate set of rules can be used.

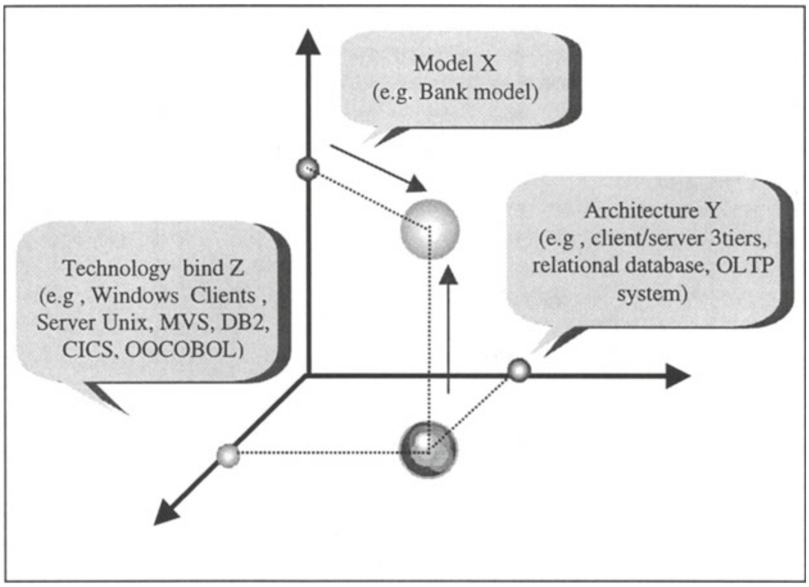

Model X
(e.g. Bank model)
Architecture Y
(e.g , client/server 3tiers, relational database, OLTP system)
Technology bind Z
(e.g , Windows Clients , Server Unix, MVS, DB2, CICS, OOCOBOL)

Figure 5. Software construction dimensions

At the moment, we have achieved a clear separation between the domain axis and the other ones. However, we still need to work on the separation of the system architecture and the software environment, which are currently too tied up in the rule scripts. In order to do that, we are working on the definition of architecture patterns that will be recognised by rules that will contain only the knowledge of how to translate a certain model and a given system architecture to a target software environment.

REFERENCES

L.Andrade and A. Sernadas, "Banking and management information system automation", Proceedings of the 13th world congress IFAC, Volume L, 1996.

Rational, Unified Modeling Language, http://www.rational.com, 1997.

J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-Oriented Modeling Technique, Prentice Hall, 1991.

The MBASE Life Cycle Architecture Milestone Package

No Architecture Is An Island

Barry Boehm, Dan Port, Alexander Egyed, and Marwan Abi-Antoun
Center for Software Engineering, University of Southern California, Los Angeles, CA 90089
{boehm, dport, aegyed, marwan}@sunset.usc.edu

Key words: Software architecture, systems architecting, architecture evaluation, model-based development, rationale capture.

Abstract: This paper summarizes the primary criteria for evaluating software/system architectures in terms of key system stakeholders' concerns. It describes the Model Based Architecting and Software Engineering (MBASE) approach for concurrent definition of a system's architecture, requirements, operational concept, prototypes, and life cycle plans. It summarizes our experiences in using and refining the MBASE approach on 31 digital library projects. It concludes that a Feasibility Rationale demonstrating consistency and feasibility of the various specifications and plans is an essential part of the architecture's definition, and presents the current MBASE annotated outline and guidelines for developing such a Feasibility Rationale.

1. ARCHITECTURE EVALUATION CRITERIA

A good software/system architecture satisfices among a number of potentially conflicting concerns. Table 1 (from Gacek et al., 1995), summarizes the major architecture-related concerns of a number of system stakeholders. These serve as a set of evaluation criteria for the architecture.

For example, the *customer* is likely to be concerned with getting first-order estimates of the cost, reliability, and maintainability of the software based on its high-level structure. This implies that the architecture should be strongly coupled with the requirements, indicating if it can meet them. The customer will also have longer-range concerns that the architecture be compatible with corporate software product line investments. *Users* need software architectures in order to be able to clarify and negotiate their requirements for the software being developed, especially with respect to future extensions to the product. The user will be interested at the architecting stage in the impact of the software structure on performance, usability, and compliance with other system attribute requirements. As with architectures of buildings, users also need to relate the architecture to their usage scenarios.

Table 1. Stakeholder concerns as architecture evaluation criteria.

Stakeholder	Concerns / Evaluation Criteria
Customer	• Schedule and budget estimation • Feasibility and risk assessment • Requirements traceability • Progress tracking • Product line compatibility
User	• Consistency with requirements and usage scenarios • Future requirement growth accommodation • Performance, reliability, interoperability, other quality attributes
Architect and System Engineer	• Product line compatibility • Requirements traceability • Support of tradeoff analyses • Completeness, consistency of architecture
Developer	• Sufficient detail for design and development • Framework for selecting / assembling components • Resolution of development risks • Product line compatibility
Interoperator	• Definition of interfaces with interoperator's system
Maintainer	• Guidance on software modification • Guidance on architecture evolution • Definition of interoperability with existing systems

Architects and systems engineers are concerned with translating requirements into high-level design. Therefore, their major concern is for consistency between the requirements and the architecture during the process of clarifying and negotiating the requirements of the system. *Developers* are concerned with getting an architectural specification that is sufficient in detail to satisfy the customer's requirements but not so constraining as to preclude equivalent but different approaches or technologies in the implementation. Developers then use the architecture as a reference for developing and assembling system components, and also use it to provide a compatibility check for reusing pre-existing components. *Interoperators* use the software architecture as a basis for understanding (and negotiating about) the product in order to keep it interoperable with existing systems. The *maintainer* will be concerned with how easy it will be to diagnose, extend or modify the software, given its high-level structure.

2. THE MBASE LIFE CYCLE APPROACH

In order to determine whether a software/system architecture is satisfactory, with respect to the criteria in Table 1, one needs considerably more than a specification of components, connectors, configurations and constraints. Considering the architecture as an island, entire of itself, puts one at a serious disadvantage in evaluating its adequacy.

We have been developing, applying and refining an approach called MBASE (Model-Based Architecting and Software Engineering) (Boehm-Port, 1998) to address this issue. It focuses on ensuring that a project's product models (architecture, requirements, code, etc.), process models (tasks, activities, milestones), property models (cost, schedule, performance, dependability), and success models (stakeholder win-win, IKIWISI (I'll Know It When I See It), business case) are consistent and mutually enforcing.

3. MBASE OVERVIEW

Figure 1 summarizes the overall framework used in the MBASE approach to ensure that a project's success, product, process and property models are consistent and well integrated. At the top of Figure 1 are various success models, whose priorities and consistency should be considered first. Thus, if the overriding top-priority success model is to "Demonstrate a competitive agent-based data mining system on the floor of COMDEX in 9 months," this constrains the ambition level of other success models (provably correct code, fully documented as a maintainer win condition). It also determines many aspects of the product model (architected to easily shed lower-priority features if necessary to meet schedule), the process model (design-to-schedule), and various property models (only portable and reliable enough to achieve a successful demonstration).

The achievability of the success model needs to be verified with respect to the other models. In the 9-month demonstration example, a cost-schedule estimation model would relate various product characteristics (sizing of components, reuse, product complexity), process characteristics (staff capabilities and experience, tool support, process maturity), and property characteristics (required reliability, cost constraints) to determine whether the product capabilities achievable in 9 months would be sufficiently competitive for the success models. Thus, as shown at the bottom of Figure 1, a cost and schedule property model would be used for the evaluation and analysis of the consistency of the system's product, process, and success models.

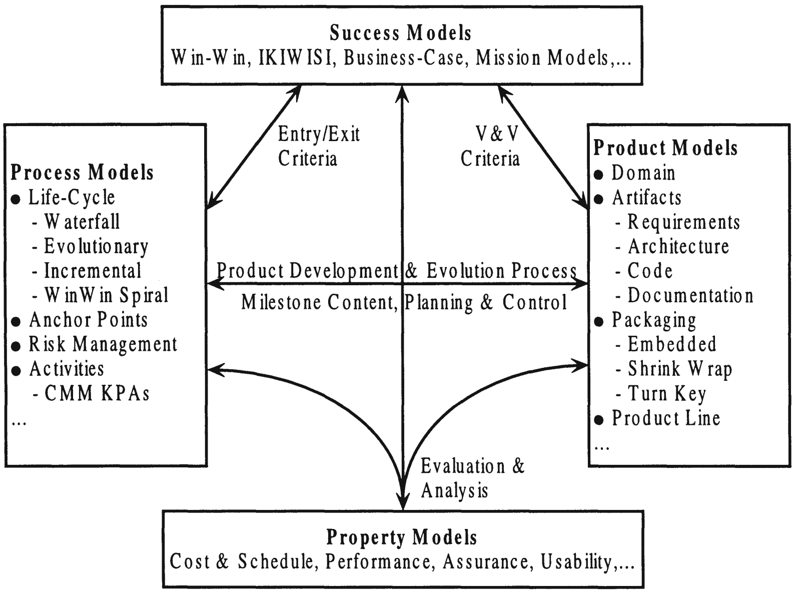


Figure 1. MBASE integration framework.

In other cases, the success model would make a process model or a product model the primary driver for model integration. An IKIWISI (I'll know it when I see it) success model would initially establish a prototyping and evolutionary development process model, with most of the product features and property levels left to be determined by the process. A success model focused on developing a product line of similar products would initially focus on product models (domain models, product line architectures), with process models and property models subsequently explored to perform a business-case analysis of the most appropriate breadth of the product line and the timing for introducing individual products.

3.1 Anchor point milestones

In each case, property models are invoked to help verify that the project's success models, product models, process models, and property levels or models are acceptably consistent. It has been found advisable to do this especially at two particular "anchor point" life cycle process milestones summarized in Table 2 (Boehm, 1996).

The first milestone is the Life Cycle Objectives (LCO) milestone, at which management verifies the basis for a business commitment to proceed at least through an architecting stage. This involves verifying that there is at least one system architecture and choice of COTS/reuse components which is shown to be feasible to implement within budget and schedule constraints, to satisfy key stakeholder win conditions, and to generate a viable investment business case.

Table 2. Content of LCO and LCA packages.

Milestone Element	Life Cycle Objectives (LCO)	Life Cycle Architecture (LCA)
Definition of Operational Concept	• Top-level system objectives and scope – System boundary – Environment parameters and assumptions – Evolution parameters • Operational concept • Operations and maintenance scenarios and parameters • Organizational life-cycle responsibilities (stakeholders)	• Elaboration of system objectives and scope by increment • Elaboration of operational concept by increment
System Prototype(s)	• Exercise key usage scenarios • Resolve critical risks	• Exercise range of usage scenarios • Resolve major outstanding risks
Definition of System Requirements	• Top-level functions, interfaces, quality attribute levels, including: – Growth vectors – Priorities • Stakeholders' concurrence on essentials	• Elaboration of functions, interfaces, quality attributes by increment – Identification of TBDs (to-be-determined items) • Stakeholders' concurrence on their priority concerns
Definition of System and Software Architecture	• Top-level definition of at least one feasible architecture – Physical and logical elements and relationships – Choices of COTS and reusable software elements – Identification of infeasible architecture options	• Choice of architecture and elaboration by increment – Physical and logical components, connectors, configurations, constraints – COTS, reuse choices – Domain-architecture and architectural style choices – Architecture evolution parameters
Definition of Life-Cycle Plan	• Identification of life-cycle stakeholders – Users, customers, developers, maintainers, interfacers, general public, others • Identification of life-cycle process model – Top-level stages, increments – Top-level WWWWWHH* by stage	• Elaboration of WWWWWHH* for Initial Operational Capability (IOC) – Partial elaboration, identification of key TBDs for later increments
Feasibility Rationale	• Assurance of consistency among elements above – Via analysis, measurement, prototyping, simulation, etc. • Business case analysis for requirements, feasible architectures	• Assurance of consistency among elements above • All major risks resolved or covered by risk management plan

* WWWWWHH: Why, What, When, Who, Where, How, How Much.

The second milestone is the Life Cycle Architecture (LCA) milestone, at which management verifies the basis for a sound commitment to product development (a particular system architecture with specific COTS and reuse commitments which is shown to be feasible with respect to budget, schedule, requirements, operations concept and business case; identification and commitment of all key life-cycle stakeholders; and elimination of all critical risk items). The AT&T/Lucent Architecture Review Board technique (Marenzano, 1995) is an excellent management verification approach involving the LCO and LCA milestones. The LCO and LCA have also become key milestones in Rational's Objectory Process or Unified Management (Rational, 1997; Royce, 1998).

4. EXAMPLE MBASE APPLICATION

4.1 Digital library multimedia archive projects

Our first opportunity to apply the MBASE approach to a significant number of projects came in the fall of 1996. We arranged with the USC Library to develop the LCO and LCA packages for a set of 12 digital library multimedia applications. The work was done by 15 6-person teams of students in our graduate Software Engineering I class, with each student developing one of the 6 LCO and LCA package artefacts shown in Table 2. Three of the 12 applications were done by two teams each. The best 6 of the LCA packages were then carried to completion in our Spring 1997 Software Engineering II class.

Table 3. Example library multimedia problem statements.

Problem Set #2: Photographic Materials in Archives

Jean Crampon, Hancock Library of Biology and Oceanography

There is a substantial collection of photographs, slides, and films in some of the Library's archival collections. As an example of the type of materials available, I would like to suggest using the archival collections of the Hancock Library of Biology and Oceanography to see if better access could be designed. Material from this collection is used by both scholars on campus and worldwide. Most of the Hancock materials are still under copyright, but the copyright is owned by USC in most cases.

Problem Set #8: Medieval Manuscripts

Ruth Wallach, Reference Center, Doheny Memorial Library

I am interested in the problem of scanning medieval manuscripts in such a way that a researcher would be able to both read the content, but also study the scribe's hand, special markings, etc. A related issue is that of transmitting such images over the network.

Project Objectives
Create the artifacts necessary to establish a successful life cycle architecture and plan for adding a multimedia access capability to the USC Library Information System. These artifacts are:

1. An Operational Concept Definition
2. A System Requirements Definition
3. A System and Software Architecture Definition
4. A Prototype of Key System Features
5. A Life Cycle Plan
6. A Feasibility Rationale, assuring the consistency and feasibility of items 1-5

Team Structure
Each of the six team members will be responsible for developing the LCO and LCA versions of one of the six project artifacts. In addition, the team member responsible for the Feasibility Rationale will serve as Project Manager with the following primary responsibilities:

1. Ensuring consistency among the team members' artifacts (and documenting this in the Rationale).
2. Leading the team's development of plans for achieving the project results, and ensuring that project performance tracks the plans.

Project Approach
Each team will develop the project artifacts concurrently, using the WinWin Spiral approach defined in the paper "Anchoring the Software Process." There will be two critical project milestones: the Life Cycle Objectives (LCO) and Life Cycle Architecture (LCA) milestones summarized in Table 1.
The LCA package should be sufficiently complete to support development of an Initial Operational Capability (IOC) version of the planned multimedia access capability by a CS577b student team during the Spring 1997 semester. The Life Cycle Plan should establish the appropriate size and structure of such a team.

WinWin User Negotiations
Each team will work with a representative of a community of potential users of the multimedia capability (art, cinema, engineering, business, etc.) to determine that community's most significant multimedia access needs, and to reconcile these needs with a feasible implementation architecture and plan. The teams will accomplish this reconciliation by using the USC WinWin groupware support system for requirements negotiation. This system provides facilities for stakeholders to express their Win Conditions for the system; to define Issues dealing with conflicts among Win Conditions; to support Options for resolving the Issues; and to consummate Agreements to adopt mutually satisfactory (win-win) Options. There will be three stakeholder roles:

- Developer: The Architecture and Prototype team members will represent developer concerns, such as use of familiar packages, stability of requirements, availability of support tools, and technically challenging approaches.
- Customer: The Plan and Rationale team members will represent customer concerns, such as the need to develop an IOC in one semester, limited budgets for support tools, and low-risk technical approaches.
- User: The Operational Concept and Requirements team members will work with their designated user-community representative to represent user concerns, such as particular multimedia access features, fast response time, friendly user interface, high reliability, and flexibility of requirements.

Major Milestones

September 16, 1996	—	All teams formed
October 14, 1996	—	WinWin Negotiation Results
October 21-23, 1996	—	LCO Reviews
October 28, 1996	—	LCO Package Due
November 4, 1996	—	Feedback on LCO Package
December 6, 1996	—	LCA Package Due, Individual Critique Due

Individual Project Critique
The project critique is to be done by each individual student. It should be about 3-5 pages, and should answer the question, "If we were to do the project over again, how would we do it better - and how does that relate to the software engineering principles in the course?"

Figure 2. Multimedia archive project guidelines.

The multimedia archives covered such media as photographic images, medieval manuscripts, Web-based business information, student films and videos, video courseware, technical reports, and urban plans. The original Library client problem statements were quite terse, as indicated in Table 3. Our primary challenge was to provide a way for the student teams to work with these clients to go from these terse statements to an LCO package in 7 weeks and an LCA package in 11 weeks.

We enabled the students and clients to do this by providing them with a set of integrated MBASE models focused on the stakeholder win-win success model; the WinWin Spiral Model as process model; the LCO and LCA artifact specifications and a multimedia archive domain model as product models; and a property model focused on the milestones necessary for an 11-week schedule (see Figure 2). Further details are provided in (Boehm et al, 1997) and (Boehm et al, 1998).

4.2 MBASE model Integration for LCO stage

The integration of these models for the LCO stage is shown in Figure 3. The end point at the bottom of Figure 3 is determined by the anchor point postconditions or exit criteria for the LCO milestone (Boehm, 1996): having an LCO Rationale description which shows that for at least one architecture option, that a system built to that architecture would include the features in the prototype, support the concept of operation, satisfy the requirements, and be buildable within the budget and schedule in the plan.

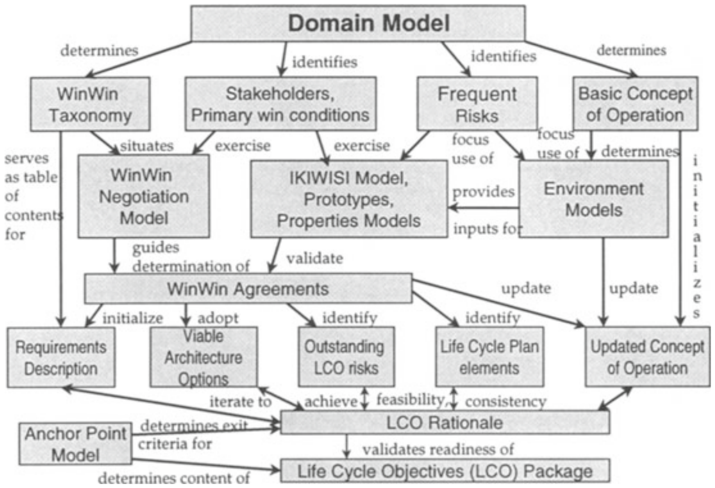
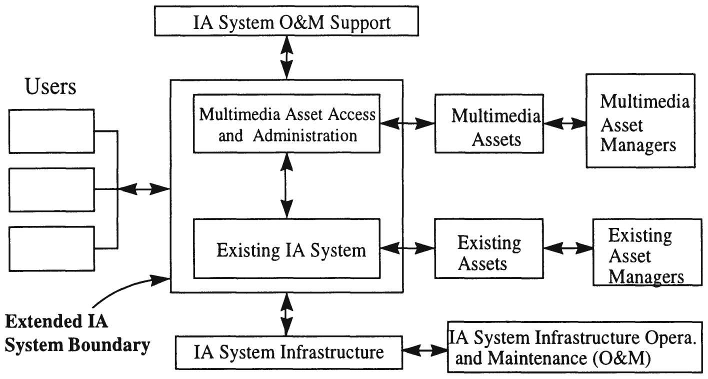


Figure 3. MBASE model integration: LCO Stage

The beginning point at the top of Figure 3 is the multimedia archive extension domain model furnished to the students, illustrated in Figure 4. The parts of the domain model shown in Figure 4 are the system boundary, its major interfaces, and the key stakeholders with their roles and responsibilities. The domain model also established a domain taxonomy used as a checklist and organizing structure for the WinWin requirements negotiation system furnished to the teams.

1. **System Block Diagram:**
   This diagram shows the usual block diagram for extensions providing access to and administration of multimedia information archive assets from an existing text-based information archive (IA) System:


IA System O&M Support
Users
Multimedia Asset Access and Administration
Multimedia Assets
Multimedia Asset Managers
Existing IA System
Existing Assets
Existing Asset Managers
Extended IA System Boundary
IA System Infrastructure
IA System Infrastructure Opera. and Maintenance (O&M)

   The system boundary focuses on the automated applications portion of the operation, and excludes such entities as users, operators, maintainers, assets, and infrastructure (campus networks, etc.) as part of the system environment. The diagram abstracts out such capabilities as asset catalogues and direct user access to O&M support and asset managers.

2. **Some Stakeholder Roles and Responsibilities**
   2.1 Asset Managers. Furnish and update asset content and catalogue descriptors. Ensure access to assets. Provide accessibility status information. Ensure asset-base recoverability. Support problem analysis, explanation, training, instrumentation, operations analysis.
   2.2-2.5 Similar roles and responsibilities defined for system operators, users, maintainers, and infrastructure service providers.

Figure 4. Multimedia archive extension domain model

As shown at the left of Figure 3, this taxonomy was also used as the table of contents for the requirements description, ensuring consistency and rapid transition from WinWin negotiation to requirements specification. The domain model also indicated the most frequent risks involved in multimedia archive applications. This was a specialization of the list of 10 most frequent software risks in (Boehm, 1989), including performance risks for image and video distribution systems; and risks that users could not fully describe their win conditions, but would need prototypes (IKIWISI).

The sequence of activities between the beginning point and the LCO end point were determined by the WinWin Spiral Model. As illustrated in Figure 5, this model emphasizes stakeholder win-win negotiations to determine system objectives, constraints and alternatives; and early risk identification and resolution via prototypes and other methods (Boehm-Bose, 1994).

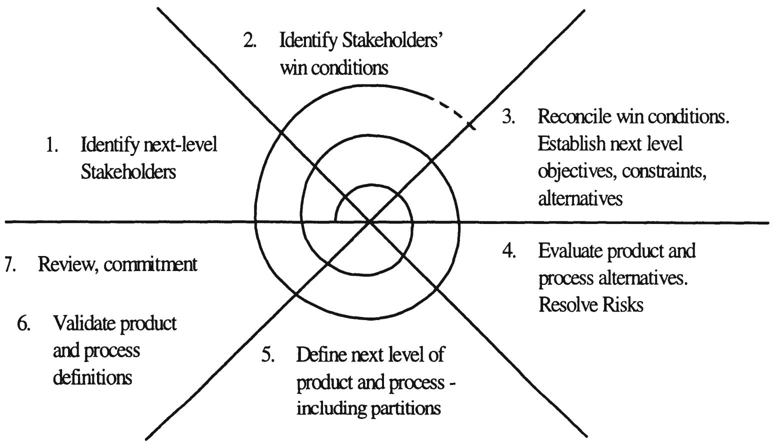

2. Identify Stakeholders' win conditions
3. Reconcile win conditions. Establish next level objectives, constraints, alternatives
1. Identify next-level Stakeholders
7. Review, commitment
4. Evaluate product and process alternatives. Resolve Risks
6. Validate product and process definitions
5. Define next level of product and process - including partitions

Figure 5. The WinWin spiral model

4.3 Project results

We were not sure how many of the 6-student teams would be able to work concurrently with each other and with their Library clients to create consistent and feasible LCO packages in 6 weeks and LCA packages in 11 weeks. With the aid of the integrated MBASE models, all 15 of the student teams were able to complete their LCO and LCA packages on time (3 of the applications were done separately by 2 teams). The Library clients were all highly satisfied, often commenting that the solutions went beyond their expectations. Using a similar MBASE and WinWin Spiral Model approach, 6 applications were selected and developed in 11 weeks in the Spring of 1997. Here also, the Library clients were delighted with the results, with one exception: an over-ambitious attempt to integrate the three photographic-image applications into a single product.

The projects were extensively instrumented, including the preparation of project evaluations by the librarians and the students. These have led to several improvements in the MBASE model provided to the student teams for Fall 1997, in which 16 teams developed LCO and LCA packages for 15 more general digital library applications. For example, in 1996, the WinWin negotiations were done before the LCO milestone, while the prototypes were done after the LCO milestone. This led to considerable breakage in the features and user interface characteristics described in the LCO documents, once the clients exercised the prototypes. As a result, one of the top three items in the course critiques was to schedule the prototypes earlier. This was actually a model clash between a specification-oriented stakeholder win-win success model and the prototype-oriented IKIWISI success model. The 1997 MBASE approach removed this model clash by scheduling the initial prototypes to be done concurrently with the WinWin negotiations.

Another example was to remove several redundancies and overlaps from the document guidelines: as a result, the 1997 LCO packages averaged 110 pages as compared to 160 in 1996. The 1997 LCA packages averaged 154 pages as compared to 230 in 1996. A final example was to strongly couple the roles, responsibilities, and procedures material in the Operational Concept Description with the product transition planning, preparation, and execution activities performed during development. Further information on the 1997-98 projects is provided in (Boehm et al., 1998). 1996-97 and 1997-98 projects can be accessed via the USC-CSE web site at http://sunset.usc.edu/classes/classes.html.

5. THE ARCHITECTURE FEASIBILITY RATIONALE AS FIRST-CLASS CITIZEN.

As indicated in Table 2, the MBASE approach treats the Feasibility Rationale as a first-class citizen in the Life Cycle Objective and Life Cycle Architecture packages. For each of the LCO and LCA components in Figure 2, we have developed an annotated outline and set of guidelines for producing the component. Below is the current version for the Feasibility Rationale.

5.1 Document overview

Why (objective): The Feasibility Rationale (FR) is the glue that holds the other components of the Life Cycle Objective (LCO) and Life Cycle Architecture (LCA) packages together. It provides evidence of the feasibility and consistency of the LCO and LCA package components.

What (content): The Feasibility Rationale includes a business case analysis demonstrating that the resources invested in the project will generate capabilities providing a satisfactory return on the investment. It also includes several satisfaction rationales addressing the various aspects of this question:

If I build the system using the given architecture and life cycle process, will it satisfy the requirements, support the operational concept, remain faithful to the key features determined by the prototype, and be achievable within the budgets and schedules in the life cycle plan?

Intended audience: The primary audiences are the LCO and LCA Architecture Review Boards. The parts dealing with client satisfaction must be understandable by the client representatives on the ARB. The technical parts must be sufficiently detailed and well-organized to enable the peers and technical experts to efficiently assess the adequacy of the technical rationale. The FR is also of considerable value to developers and other stakeholders in providing a rationale for key decisions made by the project.

Participants: The project manager is responsible for the overall content of the FR. Frequently, the business case is prepared by the author of the Operational Concept Description (OCD). Demonstrating the feasibility and consistency of portions of the LCO and LCA packages is the shared responsibility of the associated project participants. Other stakeholders may make their concurrence on win-win agreements contingent on demonstration of the agreement's feasibility in the Feasibility Rationale.

High level dependencies: The thoroughness of the Feasibility Rationale is dependent on the thoroughness of all the other LCO and LCA components. Issues incompletely covered in the Feasibility Rationale are a source of risk which should be covered in the Life Cycle Plan's (LCP) Risk Management section.

Overall tool support: Well-calibrated estimation models for cost, schedule, performance, or reliability are good sources of feasibility rationale. Others are prototypes, simulations, benchmarks, architecture analysis tools, and traceability tools (See Table 4 below for further information). The rationale capture capability in the WinWin tool is also useful.

5.2 Document outline

This section provides a table of contents for the Feasibility Rationale. Even though not all projects are alike, the people responsible for the Feasibility Rationale should consider all of these items carefully. If it is felt that some of them are not applicable, it should be noted as such for future reference. Similarly, the document outline can be expanded if there is a need. The recommended table of contents for the Feasibility Rationale document is as follows:

1. Overview
    1.1. Software Product Objectives
    1.2. Feasibility Rationale Objectives
2. Product Rationale
    2.1. Business Case Analysis
        2.1.1. Development Cost Estimate
        2.1.2. Operational Cost Estimate
        2.1.3. Estimate of Value Added and Relation to Cost
    2.2. Requirements Satisfaction
        2.2.1. Capability Requirements
        2.2.2. Interface Requirements
        2.2.3. Quality Requirements
        2.2.4. Evolution Requirements
    2.3. Operational Concept Satisfaction
    2.4. Stakeholder Concurrence
3. Process Rationale
    3.1. System Priorities
    3.2. Process Match to System Priorities
    3.3. Consistency of Priorities, Process and Resources

The following will explain in more detailed each of the items above, provide a rationale for them, show their dependencies to other sections within this document and to other documents, provide examples of their use, and give tool support recommendations whenever possible.

5.3 Document guidelines and rationale[1]

1. Overview
This section tells why the product and the plan are being developed.

1.1. Software Product Objectives
Provide a link to Section 1.1 of the Operational Concept Description (OCD). It contains a short description, in user terms, of the primary functions the product will perform, of its envisioned concept of operation, and of the user benefits expected from the product.

1.2. Feasibility Rationale Objectives
•To demonstrate that a system built using the specified architecture and life cycle process will satisfy the requirements, support the operational concept remain faithful to the key features determined by

[1] Text in **bold** can be used as is. Text in roman font indicates where project specific information needs to replace the general description provided. Text in *italic* font indicates specialization for Software Engineering I that would likely be tailored differently for other kinds of projects.

the prototype, and be achievable within the budgets and schedules in the life cycle plan.
•To rationalise development decisions in a way the prime audience (the customer and users) can understand
•To enable the customers to participate in the decision process and to express their satisfaction with the product

Integration and dependencies with other components:

- Item 1.1 is a link to the Objective items in Section 1.1 of the OCD.
- Item 1.2 may be used as is.

Additional guidelines:
None needed.

2. Product Rationale
This section furnishes the rationale for the product being able to satisfy the system specifications and stakeholders (e.g. customer, user).

2.1. Business Case Analysis
The Section describes the impact of the product in mainly monetary terms. How much does it cost to develop and to operate, how much added value does it generate, and thus how high is its return on investment. However, non-monetary factors may be also decisive. For instance, "added value" can include the improved quality of the service provided by the product.

2.1.1. Development Cost Estimate
Provide a summary of the full development cost, including hardware, software, people, and facilities costs.

2.1.2. Operational Cost Estimate
Provide a summary of the operational cost. Include also maintenance and administration cost and other costs which accumulate during transition of the product into production use (e.g. training).

2.1.3. Estimate of Value Added and Relation to Cost
Provide a summary of cost with and without the product and how much value is added by it. The value added may also describe non-monetary improvements (e.g. quality, response time, etc.) which can be critical in customer support and satisfaction. Include a return-on-investment analysis as appropriate.

2.2. Requirements Satisfaction
This section summarizes how well a system developed to the product architecture will satisfy the system requirements.

2.2.1. Capability Requirements

Show evidence that the system developed to the product architecture will satisfy the capability requirements, e.g., "capability described/demonstrated/exercised as part of included COTS component", with a pointer to the results. There is no need to restate obvious mappings from the requirements to the architecture.

2.2.2. Interface Requirements

Show evidence that the system developed to the product architecture will satisfy the interface requirements. These should include the interfaces and standards associated with the *University Computing Services* infrastructure and the *USC Integrated Library System.*

2.2.3. Quality Requirements

Show evidence that the system developed to the product architecture will satisfy the quality requirements.

2.2.4. Evolution Requirements

Show evidence that the system developed to the product architecture will satisfy the evolution requirements.

2.3. Operational Concept Satisfaction

Summarize product's ability to satisfy key operational concept elements, such as scenarios.

2.4. Stakeholder Concurrence

Summarize stakeholder concurrence by reference to WinWin negotiation results, memoranda of agreements, etc. Stakeholders may be anybody involved in the development process. For instance, a developer may claim that a certain response time cannot be achieved in a crisis mode unless nonessential message traffic is eliminated. Similarly, a customer may claim that the product does not satisfy his/her win conditions (e.g. cost). This section serves as a record of how such claims were resolved to the stakeholders' satisfaction.

Integration and dependencies with other components:

This section is highly dependent on all other documents. The cost estimates in Item 2.1 are strongly dependent on development cost (from LCP) and operational cost (from OCD). Item 2.2 maps requirements to design, which create a high dependency between the System and Software Requirements Description (SSRD), the System and Software Architecture Description (SSAD), and often the prototype. Similarly, item 2.3 creates a dependency between the OCD, the SSAD, and often the prototype. The stakeholder concurrence in Item 2.4 provides the basis for stakeholders to ratify their commitment to the project LCO and LCA packages at the ARB meetings.

Additional guidelines:

Table 4 summarizes the strengths and potential concerns for leading architecture attribute analysis methods. The rationale capture capability in the WinWin tool is also useful.

Table 4. Summary of software architecture attribute analysis methods

Method	Examples	Strengths	Potential Concerns
Current ADLs	RDD-100, StP, UML/Rose	• Static integrity (partial) • Traceability	• Dynamic integrity • Performance, cost, schedule analysis • Subjective attributes
New Generation ADLs	Rapide, Unicon, Wright	• Static, dynamic integrity • Some performance	• Model granularity and scalability • Cost, schedule, reliability, full performance • Subjective attributes
Scenario Analysis	SAAM	• Subjective attributes – Usability, Modifiability • Human-machine system attributes (partial) – Safety, security, survivability	• Largely manual, expertise-dependent • Scenario representativeness; method scalability • Verification/Validation/Accreditation • Integrity, performance, cost, schedule analysis
Simulation; Execution	Network 2.5; UNAS	• Performance Analysis • Some dynamic integrity • Some reliability, survivability	• Model granularity and scalability • Input scenario representativeness • Verification/Validation/Accreditation • Cost, schedule, subjective attributes
Parametric Modeling	COCOMO et al., Queuing Models, Reliability Block Diagrams	• Cost, schedule analysis • Reliability, availability analysis • Performance Analysis	• Subjective attributes • Static, dynamic integrity • Verification/Validation /Accreditation • Input validation

3. Process Rationale

This sections describes the rationale of the development process being able to satisfy the stakeholders (e.g. customer).

3.1. System Priorities

Summarize priorities of desired capabilities and constraints. Priorities may express time and date but also quality and others. (e.g. performance).

3.2. Process Match to System Priorities

Provide rationale for ability to meet milestones and choice of process model (e.g. anchor points in spiral model or increments, etc.).

3.3. Consistency of Priorities, Process and Resources

Provide evidence that priorities, process and resources match. E.g. budgeted cost and schedule are achievable; no single person is involved on two or more full-time tasks at any given time.

Integration and dependencies with other components:

Like the previous section, this section is also highly dependent on other documents, foremost the Life Cycle Plan (LCP) and System and Software Requirements Description (SSRD). Item 3.1 maps primarily to the capabilities in SSRD and milestones in LCP 2.2 and 2.3. Item 3.2 is a summary of LCP 4.2 which emphasis on priorities above. Item 3.3 is reasoning that the LCP is consistent and doable (especially LCP 4).

5.4 Potential pitfalls/best practices

The Feasibility Rationale is highly dependent on other components. Avoid duplicating these where mappings among components are obvious. In writing the Feasibility Rationale you should keep in mind that the primary audience is the Architecture Review Board (ARB), a mix of technical experts and general stakeholders. Portions of the FR should be tailored to the assessment needs of the various ARB members. Common pitfalls include over-reliance on vendor claims, neglect of critical off-nominal scenarios, and over-analysis of low-priority issues.

5.5 Quality criteria

The key quality criteria for the Feasibility Rationale are derived from its pitfalls. It needs to be highly consistent with the other components and it needs to be able to answer the key stakeholder questions about the feasibility of the product. It also needs to present selected system views demonstrating feasibility and consistency among the other components.

6. CONCLUSIONS

In specifying a software/system architecture, it is important not to treat the architecture as an isolated island. The architecture needs to be related to the operational concept it is supporting; the requirements the system will satisfy; the life cycle plan identifying the system's stakeholders, budgets and schedules; and any prototypes providing views of the desired system.

The satisfaction of these relationships is best recorded in a Feasibility Rationale for the architecture. For effective management review and commitment to the architecture, it is essential that the Feasibility Rationale be a first-class citizen in the architecture package. It is encouraging to note that this is so in the current draft of IEEE Standard 1471, "Recommended Practice for Architecture Description", (IEEE, 1998, Section 5.6)

REFERENCES

Boehm, B. (1989), *Software Risk Management*, IEEE-CS Press.

Boehm, B. (1996), "Anchoring the Software Process," *IEEE Software*, July, pp. 73-82.

Boehm, B. and Bose, P. (1994), "A Collaborative Spiral Process Model Based on Theory W," *Proceedings, ICSP3*, IEEE.

Boehm, B., Egyed, A., Kwan, J., and Madachy, R. (1997), "Developing Multimedia Applications with the WinWin Spiral Model," *Proceedings, ESEC/ FSE 97*, Springer Verlag.

Boehm, B., Egyed, A., Kwan, J., and Madachy, R. (1998), "Using the WinWin Spiral Model: A Case Study," *IEEE Computer*, July, pp. 33-44.

Boehm, B. and Port, D. (1998), "Conceptual Modeling Challenges for Model Based Architecting and Software Engineering (MBASE)", Proceedings, 1997 Conceptual Modeling Symposium (P. Chen, ed.), Springer Verlag (to appear)

Gacek, C., Abd-Allah, A., Clark, B.K., and Boehm, B.W. (1995), "Focused Workshop on Software Architectures: Issue Paper," *Proceedings of the ICSE 17 Workshop on Software Architecture*, April.

Garlan, D., Allen, R., and Ockerbloom, J. (1995), "Architectural Mismatch: Why Reuse is So Hard," *IEEE Software*, November, pp. 17-26.

IEEE Architecture Working Group (1998), "IEEE Recommended Practice for Architectural Description, *IEEE Std 1471, Draft Version 3.0*, 3 July.

Kazman, R., Bass, L., Abowd, G., and Webb, M. (1994), "SAAM: A Method for Analyzing the Properties of Software Architectures," *Proceedings, ICSE 16,* ACM/IEEE, pp. 81-90.

Marenzano, J. (1995), "System Architecture Validation Review Findings," in D. Garlan, ed., *ICSE17 Architecture Workshop Proceedings*, CMU, Pittsburgh, PA.

Port, D. (1998), *Integrated Systems Development Methodology*, Telos Press (to appear).

Rational (1997) *Rational Objectory Process*, Version 4.1, Rational Software Corp., Santa Clara, CA.

Royce, W.E. (1998), *Unified Software Management*, Addison Wesley (to appear).

Software Architecture at Siemens: The challenges, our approaches, and some open issues

Lothar Borrmann and Frances Newbery Paulisch
Siemens AG, Corporate Technology, Software and Engineering,
D-81730 Munich, Germany.
Lothar.Borrman@mchp.siemens.de, Francis.Paulisch@mchp.siemens.de

Key words: Software architecture, software process, frameworks, architecture assessment, product families, evolution of software architectures

Abstract: The importance of software architecture in the design of large software systems is unquestioned in both the academic and industrial software engineering communities. At Siemens, software is an important, often dominant, factor in the success of our products and this trend towards software is increasing as software becomes even more prevalent in our product spectrum. Our experience indicates clearly that attention to three aspects - to people, to process, and, in particular, to architecture - are important for successful product developments. This paper lists some of the challenges that we face in the area of software architecture, what approaches we have taken as well as a set of issues that require further attention in future.

1. INTRODUCTION

The importance of software architecture in the design of large software systems is unquestioned in both the academic and industrial software engineering communities. In the past few years, there has been a growing body of good literature (e.g., Bass, Clements and Kazman, 1997, Garlan and Shaw, 1996, Jacobson, Griss and Jonsson, 1997, Kruchten, 1995, Perry, 1997) on the topic of software architecture as well as conferences and workshops focusing on this topic. At Siemens, we welcome the efforts, such as this Working IFIP Conference on Software Architecture, to provide a forum for practicing software architects to exchange information with the academic and research community in this area. Practicing software architects are, generally, very busy people because their skills are so important and are in such high demand. We hope that this event will help make it possible for practicing architects to learn as quickly and effectively as possible what techniques are currently available and what techniques will soon be sufficiently mature to be applied.

Siemens is a large, globally-operating, electrical engineering and electronics company with a very diverse range of products. Siemens consists of about a dozen groups that cover the core business areas:

- energy (e.g., power plants)
- industry (e.g., industrial plants)
- communication (e.g., switching systems and mobile phones)
- information (e.g., software products)
- transportation (e.g., control systems for trains and cars)
- health care (e.g., medical imaging equipment)
- components (e.g., ASICs)
- lighting

Global development (e.g., geographically distributed and culturally diverse teams all working on one project) and global sales (e.g., country-specific customization of products) play a significant role in the product development of software, systems, and industrial plants at Siemens. Some of our products are highly customized individual solutions; others are more oriented towards the mass market. Many of our products have strong requirements in the areas of safety, reliability, robustness, and performance and this is further complicated by the fact that this often has to be achieved in real time. Furthermore, the maintainability and serviceability (sometimes over decades!), as well as the evolution of our products is important for our businesses.

Due to the nature of our products, many of the Siemens groups have had a strong orientation towards hardware, electrical engineering, or mechanical engineering. But software is increasingly becoming an important, often dominant, factor in the success of their products and this trend towards software seems to be increasing more rapidly all the time. To give you a feeling for the importance of software at Siemens, consider that:

- More than 50% of our enterprise-wide sales stem from software-based products or systems.
- 27,000 software engineers are employed worldwide (about 10% of our employees).
- Some of our projects are very large, global projects, e.g., one with 2000 developers in 13 countries.

We see software as the key to being able to meet the challenges of flexibility, time-to-market, and reducing costs while maintaining quality.

Because of the importance of software for our company an enterprise-wide "software initiative" was founded in 1995 as part of the "top" initiative (Gonauser, 1997). A dozen groups (e.g., automation, automotive, train transportation, health care, communication, etc.) as well as regional units like Siemens Austria in Vienna and Siemens Switzerland in Zurich actively participate in the enterprise-wide software initiative. One of our main goals of this initiative is to promote an intensive and extensive exchange of information in regards to people, process, and architecture within the various Siemens groups, so that we can learn from each other and improve and help maintain our software expertise. The software initiative focuses on particular topics that are most important to their businesses, e.g., cycle time, cost, quality, process innovation, architecture evolution, measurement-based management, component-oriented development, etc.

The authors have a good overview of the challenges facing our business groups in the areas of software, in particular in the areas of software architecture, software processes, and the human factors involved in both. This paper lists some of the challenges that we face (especially in the area of software and systems architecture), what progress we have made thus far, and what areas we intend to focus on in the future.

2. PEOPLE, PROCESS, ARCHITECTURE

Our experience indicates clearly that attention to all three aspects (people, process, and architecture) are important for successful product developments. Of course, there are many areas where process, people, and software are intertwined, e.g., in our increased focus on component-oriented development, we need to concentrate, not only architectures, but also the processes, and people that support this approach. It is a well-known fact that the capabilities and motivation of the people involved in software development can and does vary widely. Having clearly defined processes and architectures is an important factor both directly and indirectly (via the thereby satisfied customers) in employee satisfaction and motivation.

One of our central research and development departments has a long history (since 1993) of performing process and architecture assessments and associated improvement projects. In the past five years, this group has conducted about 100 process assessments and 10 architecture assessments (Mehner et al., 1998). The main objectives of the assessments are:

- to analyze and evaluate processes and architectures. Architectures are as important as processes for optimizing the triad "costs - quality - schedule" and simultaneously being flexible enough for introducing new and innovative products in the market in time.

- to identify in detail the potential to start and perform improvement projects.
- to specify in detail measurements to improve the evaluated process and architecture and realize this potential.

We place significant emphasis on measurement and evaluation, because our belief is that you can only effectively improve what you can measure. To get optimal insight into the process capabilities of an organization, various interrelated measurement and evaluation techniques have to be applied. In order to drive changes in software and engineering processes, it is necessary to set precisely defined goals and measure progress towards these goals. These goals may be at the project, process, or business level. At Siemens, we have developed and use the so-called top^{Six} controlling instruments (Lebsanft and Rheindt, 1998) for the software-related business (this includes metrics and controlling instruments at the process, project, and management level). These allow us to control the six success factors:

1. customer satisfaction
2. quality
3. cycle time
4. productivity
5. process maturity
6. technology maturity

These final two, process maturity and technology maturity, are where architecture aspects are relevant. As in the capability maturity model (CMM) of the Software Engineering Institute, the process maturity measurement includes a focus on design/architecture issues (e.g., to what extent are architecture reviews performed, how architectures are embedded in the product line management, etc.). The technology maturity indicates the importance of technological trends and how well an organization can adapt to them for business benefit. One aspect of this is related to architecture issues (e.g., the use of COM/DCOM, CORBA, Java, etc.).

Tailored to the needs of the various business groups, the top^{Six} aim to give an objective appraisal of the current state and to allow the early detection of changes. Furthermore, interpreted together, they provide a good understanding of the capability of the organization to develop software. Several of the business groups already have extensive experience with these and other measures and can report on their effectiveness, especially in relation to controlling and improving their development processes.

2.1 Architecture assessments

Due to the increased complexity and size of today's systems, an increased focus on the software architecture of a system is needed. The architecture gives the overall structure of the system and identifies the main components and their interactions. The long-term success of a system depends in large part on the quality of the software architecture. At Siemens, we have developed a method called "system architecture analysis" (SAA) (Gloger et al. 1997) that allows us to evaluate the major technical concepts of an architecture as well as to serve as the basis for proposed improvements for the evolution of the architecture. It includes the determination of the evaluation and weighting of the criteria for the architecture, an analysis of the design decisions, an analysis of the interdependencies between the design decisions, and the evaluation of this information that makes the pros and cons of the various design alternatives clearer (see *Figure 1*).

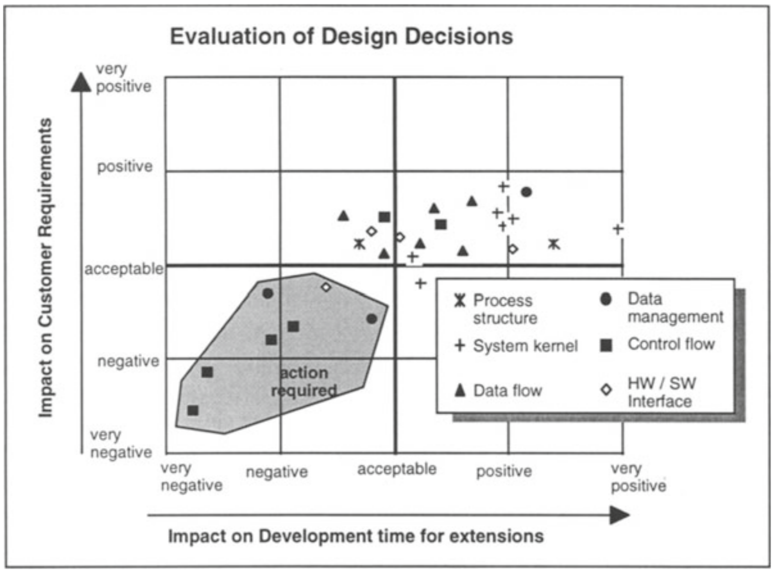


Figure 1: Making design alternatives clearer and indicating where action is required

In comparison to the "Software Architecture Analysis Method" (SAAM) (Kazman et al., 1994), our approach is narrower and more focused on the immediate needs of our business groups.

3. EXPERIENCE AND LESSONS LEARNED

The following section describes some of our experiences and lessons learned in the area of software architecture at Siemens.

3.1 Use innovative processes

We have found that being willing and able to use more innovative processes that, for example, take the purchasing of commercial off-the-shelf software and standard components into account and/or that allow regular and incremental development (e.g., weekly builds) to be effective. Results on the order of 50% reduction in cycle time and 35% reduction in development costs over a period of about five years are not uncommon. An additional advantage is the ability to meet customer requirements and/or react to our competition more quickly. Some organizations are moving from a product-oriented to a process-oriented development approach and the initial results here are very positive, both for the development costs, accuracy at meeting the deadline, as well as for the motivation of the staff.

3.2 Migrate from software "construction" to "composition"

The migration from construction to composition has a strong effect on both the process and the architecture. It is becoming more common to at least consider the integration of existing components (either our own or third-party commercial off-the-shelf (COTS) products). This is considered for various reasons, for example, to focus on our core areas of technical expertise, to reduce costs, to enhance productivity, to adhere to standards, etc. It is, however, very important to be aware both of the potential problems due to architectural mismatch (Garlan, Allen and Ockerbloom, 1995) and the potential process-related challenges associated with the integration of components in a product, for example:

- In order to be flexible and meet the needs of a large number of potential users, the components may be slower and larger than components developed to more closely match the actual needs
- Changes are often impossible (e.g., "black box" components) or difficult (because one has to understand the architecture to change it)
- There are complicated legal and contractual issues, such as liability, associated with this approach.

Typically, we have found that incremental processes work better for composition because they allow the integration of the "foreign" components earlier in the development and because they allow for the early analysis of key performance issues. Not only a good process, but also a clear software architecture, is necessary for this approach to work well.

3.3 Architecture review sessions are effective

At least one of our business groups regularly holds architecture review sessions in which a real architecture is presented and discussed in detail. This has been shown to have advantages both for those directly involved in the architecture, because they receive valuable suggestions for improvement, and for the other participants who profit from a better understanding of the architecture.

3.4 Frameworks are useful for both process and architecture

We have found that frameworks are a key for achieving an optimal balance between stability and flexibility for both development processes as well as for the software architecture. For example (Völker and Wackerbarth, 1997) cycle time was reduced in the digital switching system area by 50% in the past five years due, to a large extent, on the structure of their development processes. It provides a globally agreed-upon "process framework" giving the stable structure and within the "process components." There is a significant amount of flexibility allowed for use in the specific business groups.

Similarly, we have found a framework approach for software architecture to be effective for software development. In our experience, although there is a significant investment that has to be made in such a framework up front, in the long term the return on investment can be substantial. In three of our business groups we have been able to show that approaches based on components, design patterns (Gamma et al. 1995, Buschmann et al. 1996, Beck et al. 1996), and frameworks have led to significant reductions in cost and faster and more accurate time-to-market (i.e., that the investment in the architecture has started to pay off). In one business group the total software development costs that had been increasing by 30–35% annually have now actually decreased by 10% annually. This reduction is due to their architecture-based approach and their investment in components, design patterns and frameworks, despite similar time constraints and requirements.

As cited in (Buschmann et al., 1998), the decision as to whether a framework approach for software architecture is worthwhile depends on a number of factors, for example, the stability of the subject area and the technology. An in-depth analysis should be done beforehand to decide whether the investment is likely to pay off in the long run. Note also that the benefit is not just in the reduced development time of the nth framework; the stable framework is likely to be better understood and more tested (i.e., usually of higher quality) than a new development.

The stability and flexibility given by the framework approach is also particularly useful in our global developments since it allows a clear definition of the interactions involving the geographically-distributed people, the processes, and the architecture.

3.5 Investment in domain analysis/product family/product line can be worthwhile

Having a good software architecture is the key to building systems that are scaleable and configurable and thus can be used effectively for a product family or product line. We have had, for example, the case where, for historical reasons, two independent product families had been developed and it became clear that it would make business sense to merge them. We applied our so-called "Harmonization of Software Architectures and Platforms (HAP)" process (Gloger et al., 1998) to harmonize these heterogeneous product spectrums (see *Figure* 2).

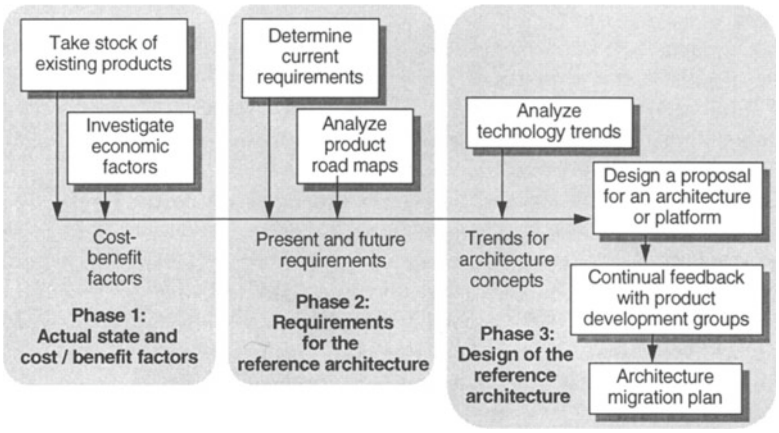

Take stock of existing products
Investigate economic factors
Cost-benefit factors
Phase 1: Actual state and cost / benefit factors
Determine current requirements
Analyze product road maps
Present and future requirements
Phase 2: Requirements for the reference architecture
Analyze technology trends
Design a proposal for an architecture or platform
Trends for architecture concepts
Continual feedback with product development groups
Phase 3: Design of the reference architecture
Architecture migration plan

Figure2: The harmonization of software architectures and platforms (HAP) process

The basic principles of this process are to:

– Determine the current state (i.e., what products have harmonization potential) and the cost-benefit factors. This is done in a set of workshops that includes both the development teams and the marketing and sales departments.

- Determine the requirements of the reference architecture. This is a list of requirements together with a set of priorities indicating their significance.
- Define the reference architecture. The aforementioned SAA method (Gloger et al., 1997) is used here to help structure the alternatives and make the decision-making process more transparent. As part of this phase, we also develop a plan for the migration to the new reference architecture.

Note that, although the harmonization approach is of benefit in the product development phase, the benefits further down the line, especially in the logistics (e.g., the installation, commissioning, maintenance, and service phases of the product) are even more dramatic. Configuring, delivering, and installing a dozen different versions of a product is time-consuming and error-prone; architecture harmonization helps reduce the number of alternatives.

From the business perspective it is also important to note that such a harmonization approach can only work if the development processes are also changed accordingly. If multiple units within an organization focus only on their own cost/benefit scenarios, it is very difficult to get them to support such a merged approach. Certainly, attention to the process implications of the product line approach is essential (see also Perry, 1996).

3.6 N-tier architectures are popular, especially for distributed systems

4-tier (user interface, web-top server, application server, database/network) or n-tier architectures are used increasingly especially because the browser-based interface offers increased platform-independence. This approach is particularly popular for large distributed systems and IT systems; for example, the "ComUnity" approach of Siemens Nixdorf Information Systems is based on such a structure. This approach can also be very suitable for mobile devices.

3.7 Maintain an online repository of "best practices"

Several groups have established centers of particular technical competence within their groups and have found this approach to be effective. The software initiative has begun extending this notion by encouraging so-called "best practice networking" in which key staff members serve as "champions" of a particular topic area in our online information repository.

4. PROBLEMS FACED BY SOFTWARE ARCHITECTS IN INDUSTRY

Although section 3 lists some of the areas where our architecture approaches are effective, there are a great deal of unsolved or insufficiently solved problems in the global software architecture community. In this section we list some software architecture problems we encountered in professional software projects within our company. We hope that the discussions at the WICSA1 working conference will help find ways to address these issues.

4.1 Increasing system complexity

With the move from monolithic systems to client-server architectures and n-tier systems, system complexity was reduced by cutting the system into pieces. At the same time, complexity was increased by introducing distribution and heterogeneity.

a) To cope with distribution, communication layers and middleware platforms were added which are not always understood by the average programmer.
b) The interworking of different operating systems, GUIs, database systems, middleware platforms, etc. imposes a number of technical difficulties and requires a combined expertise which is rarely found in a single software architect.
c) "Standard" communication schemes and interfaces are developing rapidly, causing incompatibility issues and necessitating continuous updating of the system components.
d) Programming environments, testing/monitoring tools, and even most conventional programming languages are designed for monolithic systems, but support for distributed environments is still rudimentary.
e) Designing complex systems requires design methods which are sound, but lead to tangible results in a reasonable time. The use of design patterns is one example, but still, architecture design is seen as more of an "art" and the architectural design process is not well-defined.

4.2 Architecture of high-lifetime and rapidly evolving systems

In our company, there exist systems that have a system lifetime of 30 years, like railway control systems. Other systems have a considerable product life time, with continuous development according to technical progress, like public telephone switching systems. Systems with a high lifetime and/or those that evolve rapidly pose the following problems:

a) Architectural drift is a well-known issue. A properly defined architecture is being continuously degraded by modifications and added functionality. As designers and architects move within the organization, the knowledge about the original architecture fades away, and developers are not capable of preserving the proper architecture when making changes. To improve preservation of architectural knowledge, better ways to document architectures - in the large - are desirable.

b) Once the above has happened, the system with its degraded architecture is considered a "legacy system". It is a common approach to avoid changes to legacy systems, and rather complement them with new components when functionality is to be added. Here we need standard approaches for ensuring interoperability of old systems with new systems. One example is the use of wrappers for legacy systems.

c) A well-designed architecture must support change - it must be stable to allow flexibility of the systems which are built after this architecture. This includes system scalability, in order to be able to provide a family of systems according to user needs, but also the potential to fulfill new requirements, interoperate with other systems, or adapt to technological changes. To achieve this, assessment schemes for architectural quality are required.

4.3 Issues caused by organizational structure

In large companies like Siemens, a clear organizational structure is required for business needs. There is a strong trend to enforce the separation of business units, product lines, etc. While this is a commercial necessity, it can impede the enforcement of an architectural strategy:

a) When developing systems in a vertical structure, each system architecture is typically designed independently. Different approaches may be used and different platforms may be chosen. Interoperability and scalability are in question, and support for various entirely different systems may be required, with high development and maintenance cost incurred. In such a situation, a harmonization of these architectures (as described in section 3) may help, but this is a non-trivial task.

b) To avoid the problem in an early stage, it is recommended to enforce an architecture strategy across organizational boundaries. A way to achieve this is to install an architecture group that works closely with the system designers. In a few cases, we have seen the cooperation of system designers with architects fail. Architects did not have the power or the acceptance to enforce a valid strategy, and were made redundant in the end. Best practice studies are required to find the correct approach for the empowerment of the architects and the way to cooperate with designers.

c) When the focus of a design and development team is on short-term commercial success, a "quick and dirty" approach may be favored over a properly designed system architecture. While this may lead to a short-term success, it may be costly in the long run. We must find ways to properly count the value of an architecture as an investment. As an example, consider the framework approach: As a rule of thumb, a framework needs to be used three times before the cost of its design and implementation is offset by the reduction in system development cost. Framework development is thus an investment.

4.4 Architectures including COTS components or platforms

It is a common trend in system development projects not to develop all software from scratch, but rather include commercial off-the-shelf (COTS) software components in the system. When we talk about software components, we do use the term in the narrow sense of component-based software; we mean that certain parts of the software system are obtained from commercial software suppliers as standard products. Example of such components include operating systems, database systems, and middleware platforms. In a modern software development environment the portion of COTS software in systems being developed is steadily increasing.

The goal of using COTS software is clear: Development cost and time-to-market is reduced, system functionality is improved, well-proven components are expected to have far fewer programming errors than newly developed code, and the component supplier is expected to take care of component maintenance. Nevertheless, these goals are not always met and we have seen some COTS-based development projects fail; why?

a) Traditional software engineering methods and development processes do not take COTS components into account. A top-down design approach will usually not lead to an architecture which fits the components available, rather a mixture of top-down and bottom-up is required. In the development process, a strong coupling of the requirements engineering stage and the design stage, incorporating rapid prototyping steps, are required, as the properties of the COTS components may strongly influence the properties of the resulting system. Working with COTS components is still more a matter of professional software-engieering experience than something that is well-understood and taught in courses.

b) Software components are more complex than nuts and bolts. Their behavior is—in the best case—only partially documented. The behavior of such a component in a given environment, where it interoperates with custom software and—worse—with other COTS software components, is often unpredictable. Suppliers of COTS components will in most cases give no guarantee that certain non-functional requirements will be met. Moreover, COTS components are not tailored for the specific purpose they are needed for. As a result, they provide unnecessary functionality, at the cost of increased resource consumption and degraded performance. Quality assurance techniques for software components are required.
c) Software development effort is now substituted with different tasks: Selection of components and suppliers, quality assurance and, often, negotiation of licenses. These "new" tasks require different skills and the techniques for implementing them is not widely established yet. When component evaluation and contract negotiation are required, a considerable amount of time may be spent—perhaps even more than the time saved in the reduced development time.
d) When a system is built with a COTS software platform as the base, there is a high risk that the system will be tailored to this platform, which leads to the so-called "vendor lock-in" problem. When this occurs, the system is strongly dependent on the base platform and its supplier. This is a considerable commercial and technological risk. When the platform is not available any more, for whatever reason, a major redesign of the system can be expected. Platform updates may require costly modifications to the overall system. Measures to reduce the risk, like the provision of an isolation layer, are not always appropriate, or may not be chosen, for example, due to the additional overhead and complexity. It is also well known that certain software vendors use this issue to improve their position in the market.

The above shows that the use of COTS components has its price, and that there are quite a few open questions. For some of them, research is under way, but solutions are not widely established yet.

5. OUTLOOK

In addition to the open issues mentioned in section 4, our company is particularly interested in the following issues:

a) What are the interrelationships between product families and process families (e.g., as discussed in Sutton and Osterweil, 1996)?
b) How can we ensure the quality of the individual components?

a) What can we foresee about the quality for software architectures built out of components? In other words, if we use a set of components with particular non-functional properties (e.g., very robust and another very safe) are there any conclusions we can make about the non-functional properties of the whole as opposed to the parts (e.g., as discussed in Clements et al., 1995)?
b) What techniques have proven most effective for the evolution of software architectures? We want to continue to build architectures that are easy to upgrade and that maybe even can be so "clever" as to adapt to particular configurations without human intervention.
c) Software architects are very busy and rightly so, since their knowledge is of great value to the company. How can we make sure that they have enough time not only to work on project-specific issues, but to stay up-to-date with current and future directions.

Within Siemens, both the software initiative and in the projects we have been involved in, we have seen that the exchange of information in these areas can be very beneficial. We hope that by further increasing the interaction with the international software engineering community and sharing some of our experiences that we can provide even more valuable information to our practicing software architects.

ACKNOWLEDGEMENTS

We gratefully acknowledge the discussions and valuable hints provided by Michael Stal and Axel Völker of Siemens that helped improve this paper.

REFERENCES

Buschmann,F; Meunier, R; Rohnert, H.; Sommerlad, P; Stal, M. (1996), *Pattern-oriented Software Architecture - A System of Patterns*, John Wiley.

Beck,K; Coplien, J; Crocker, R; Meszaros, G; Paulisch, F; Vlissides, J. (1996), *Industrial Experience with Design Patterns*, Proceedings of ICSE-18, Berlin, Germany.

Bass, L.; Clements, P.; Kazman, R. (1997), *Software Architecture in Practice*, Addison-Wesley.

Buschmann, F; Geisler, A.; Heimke, T.; Schuderer, C. (1998), *Framework-Based Software Architectures for Process Automation Systems,* 9th IFAC Symposium on Automation in Mining, Mineral, and Metal Processing, Cologne, Germany.

Clements, P; Bass, L.; Kazman, R.; Abowd, G. (1995), *Predicting software quality by architecture-level evaluation*, Proceedings of the 5th Intl. Conference on Software Quality, Austin.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (1995), *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Garlan, D.; Allen, R.; Ockerbloom, J. (1995), *Architectural Mismatch: Why Reuse Is So Hard*, IEEE Software.

Garlan, D.; Shaw, M. (1996), *Software Architecture - Perspectives on an Emerging Discipline*, Prentice-Hall.

Gloger, M.; Jockusch, S.; Weber, N. (1997), *Assessment and Optimization of System Architectures - Experience from Industrial Applications at Siemens*, Proceedings of the European Software Engineering Process Group Conference, Amsterdam, Netherlands.

Gloger, M.; Jockusch, S.; Weber, N. (1998), *Harmonisierung von Software-Architekturen und Plattformen (HAP): Erfahrungen aus dem industriellen Kontext bei Siemens* (in German), Proceedings of the Softwaretechnik '98 Conference, Paderborn, Germany.

Gonauser, M. (1997), *Mit einer Software-Initiative zur Weltspitze* (in German), Computerwoche.

Jacobson, I.; Griss, M.; Jonsson, P. (1997), *Software Reuse: Architecture, Process, and Organization for Business Success*, Addison-Wesley.

Kazman, R.; Bass, L.; Abowd, G.; Webb, G. (1994), *SAAM: A Method for Analyzing the Properties of Software Architectures*, Proceedings of the ICSE-16, Sorrento, Italy.

Kruchten, P. (1995), The 4+1 View Model of Architecture, IEEE Software.

Lebsanft, K.; M. Rheindt (1998), *Improvement of the development to increase customer satisfaction*, Proceedings of the Federation of Software Metrics Associations in Europe (FESMA), Antwerpen, Belgium.

Mehner, T.; Messer, T.; Paul, P.; Paulisch, F.; Schless, P.; Völker, A. (1998), *Siemens Process Assessment and Improvement Approaches - Experiences and Benefits,* Proceedings of the COMPSAC '98 Conference, Vienna, Austria.

Perry, D. (1996), *Product Line Implications for Process*, 10th Intl. Software Process Workshop, Ventron, France.

Perry, D. (1997), *State of the Art in Software Architecture*, Proceedings of the ICSE-19, Boston.

Sutton, S.; Osterweil, L. (1996), *Product Families and Process Families*, 10th Intl. Software Process Workshop, Ventron, France.

Völker, A.; Wackerbarth, G. (1997), *Competence in Software and Engineering: Siemens Software Initiatives*, Proceedings of the European Software Engineering Process Group Conference, Amsterdam, Netherlands.

Architectural Design to Meet Stakeholder Requirements

L. Chung[1], D. Gross[2] & E. Yu[2]
[1]Computer Science Program, University of Texas, Dallas, USA &
[2]Faculty of Information Studies, University of Toronto, Toronto, Ontario, Canada
chung@utdallas.edu, {gross, yu}@fis.utoronto.ca

Key words: Software architecture, rationale, stakeholders, organization modeling, requirements, quality attributes, architectural properties, non-functional requirements, process-oriented, softgoal, satisficing, design reasoning

Abstract: Architectural design occupies a pivotal position in software engineering. It is during architectural design that crucial requirements such as performance, reliability, costs, etc., must be addressed. Yet the task of achieving these properties remains a difficult one and it is made even more difficult with the shift in software engineering paradigm from monolithic, stand-alone, built-from-scratch systems to componentized, evolvable, standards-based, and product line oriented systems. Many well-established design strategies need to be reconsidered as new requirements such as evolvability, reusability, time-to-market, etc., become more important. This paper outlines an approach that formulates architectural properties such as modifiability and performance as "softgoals" which are incrementally refined. Tradeoffs are made as conflicts and synergies are discovered. Architectural decisions are traced to stakeholders and their dependency relationships. Knowledge-based tool support for the process would provide guidance during design as well as records of design rationales to facilitate understanding and change management.

1. INTRODUCTION

The importance of architectural design is now widely recognized in software engineering, as evidenced by the recent emergence of seminal reference texts e.g., (Shaw & Garlan, 1996; Bass, 1998) and several international workshop series and special sessions in major conferences. It is acknowledged, however, that many issues in software architecture are just beginning to be addressed. One key task that remains a difficult challenge for practitioners is how to proceed from requirements to architectural design.

This task has been made much more difficult as a result of today's changing software environment. Systems are no longer monolithic, built from scratch, or operate in isolation. Systems built in the old paradigm have contributed to the legacy system problem. Today's systems must be developed quickly, evolve smoothly, and interoperate with many other systems. Today's architects adopt strategies such as reusability, componentization, platform-based, standards-based, etc., to address new business level objectives such as rapid time-to-market, product line orientation, and customizability. Two important aspects may be noted in this shift in software engineering environment:

1. There have been significant shifts in architectural quality objectives.
2. Architectural requirements are originating from a much more complex network of stakeholders.

System-wide software qualities have been recognized to be important since the early days of software engineering. For example, (Boehm, 1976) and (Bowen, 1985) classified a number of software attributes such as flexibility, integrity, performance, maintainability, etc. It is well known that these quality attributes (also referred to as non-functional requirements) are hard to deal with, because they are often ill defined and subjective. The recent flurry of activities on software architecture involving researchers and practitioners have refocused attention on these software qualities since it is realized that system-wide qualities are largely determined during the architectural design stage (Boehm, 1992; Perry, 1992; Kazman, 1994; Shaw & Garlan 1996; Bass, 1998). With the shift to the new, fast-cycled, component-oriented software environment, priorities among many quality objectives have changed, and new objectives such as reusability and standards compliance are becoming more prominent. While performance will continue to be important, it must now be traded off against many kinds of flexibility. As a result, many architectural solutions that were well accepted in the past need to be rethought to adapt to changes in architectural objectives.

When systems were stand-alone and had definite lifetimes, requirements could usually be traced to a small, well-defined set of stakeholders. In the new software environment, systems tend to be much more widely interconnected, have a more varied range of potential customers and user groups (e.g., due to product line orientation), may fall under different organizational jurisdictions (at any one time, and also over time), and may evolve indefinitely over many incarnations. The development organization itself, including architects, designers, and managers, may undergo many changes in structure and personnel. Requirements need to be negotiated among stakeholders. In the case of architectural quality requirements, the negotiations may be especially challenging due to the vagueness and open-endedness of initial requirements. Understanding the network of relationships among stakeholders is therefore an important part of the challenge faced by the architect practitioner.

These trends suggest the need for frameworks, techniques, and tools that can support the systematic achievement of architectural quality objectives in the context of complex stakeholder relationships.

In this paper, we outline an approach that provides a goal-oriented process support framework, coupled with a model of stakeholder relationships. The paper includes simplified presentations of the NFR Framework (Chung, 1998) and the *i** framework (Yu, 1995). A web-based information system example, incorporating a KWIC component, is used to illustrate the proposed approach.

2. GOAL-ORIENTED PROCESS SUPPORT FOR ARCHITECTURAL DESIGN

Consider the design of a web-based information system. There would be a set of desired functionalities, such as for searching information, retrieving it, scanning it, downloading it, etc. There would also be a number of quality requirements such as fast response time, low storage, ease of use, rapid development cycle, adaptability to interoperate with other systems, modifiability to offer new services, etc. The functional side of the requirements are handled by many development methodologies, from structured analysis and design, to recent object-oriented methods. Almost all these methods, however, focus overwhelmingly, if not exclusively, on dealing with functional requirements and design. While there is almost universal agreement on the crucial importance of achieving the quality requirements, current practice is often ad hoc, relying on after-the-fact evaluation of quality attributes. Techniques for evaluating and assessing a completed architectural design ("product") are certainly valuable. However, such techniques usually do not provide the needed step-by-step ("process") guidance on how to seek out architectural solutions that balance the many competing requirements.

Complementary to the product-oriented approaches, the NFR Framework (Chung, 1993, 1998) takes a *process-oriented* approach to dealing with quality requirements. In the framework, quality requirements are treated as (potentially conflicting or synergistic) goals to be achieved, and used to guide and rationalize the various design decisions during the system/software development. Because quality requirements are often subjective by nature, they are often achieved not in an absolute sense, but to a sufficient or satisfactory extent (the notion of *satisficing*). Accordingly, the NFR Framework introduces the concept of *softgoals*, whose achievement is judged by the sufficiency of contributions from other (sub-) softgoals. Throughout the development process, consideration of design alternatives, analysis of design tradeoffs and rationalization of design decisions are all carried out in relation to the stated softgoals and their refinements. A *softgoal interdependency graph* is used to support the systematic, goal-oriented process of architectural design. It also serves to provide historical records for design replay, analysis, revisions, and change management.

For the purpose of illustration, let us consider a small part of the example in which a keyword in context (KWIC) system is needed. The KWIC system is part of a web information system, used to support an electronic-shopping catalog. Suppose the KWIC system architect is faced with an initial set of quality requirements: "the system should be modifiable" and "the system should have good performance". In the aforementioned process-oriented approach, the architect explicitly represents each of these as a softgoal to be achieved during the architectural design process. Each softgoal (e.g.,, Modifiability [system]) is associated with a type (Modifiability) and a topic (system), along with other information such as importance, satisficing status and time of creation. Figure 1 shows the two softgoals as the top level nodes.

As these high level requirements may mean different things to different people, the architect needs to first clarify their meanings. This is done through an iterative process of softgoal refinement which may involve reviewing the literature and consulting with domain experts. After consultation, the architect may refine Modifiability [System] into three offspring softgoals: Modifiability [Algorithm], Modifiability [Data representation], and Modifiability [Function]. This refinement is based on topic, since it is the topic (System) that gets refined, while the softgoal type (Modifiability) is unchanged. This step may be justified by referring to the work by Garlan and Shaw (Garlan, 1993), who consider changes in processing algorithm and changes in data representation, and to Garlan, Kaiser, and Notkin (Garlan, 1992), who extend the consideration with enhancement to system function. Similarly, the architect refines Performance [System], this time based on its type, into Space Performance [System] and Time Performance [System], referring to work by Nixon (Nixon, 1993).

Figure 1 shows the two refinements. In the figure, a small "arc" denotes an "AND" *contribution*, meaning that in order to satisfice the parent softgoal, all of its offsprings need to be satisficed. As will be shown later, there are also other contribution types, including "OR" and partial positive

(+) or negative (-) contributions. Contribution types are important for deciding the satisficing status of a softgoal based on contributions towards it.

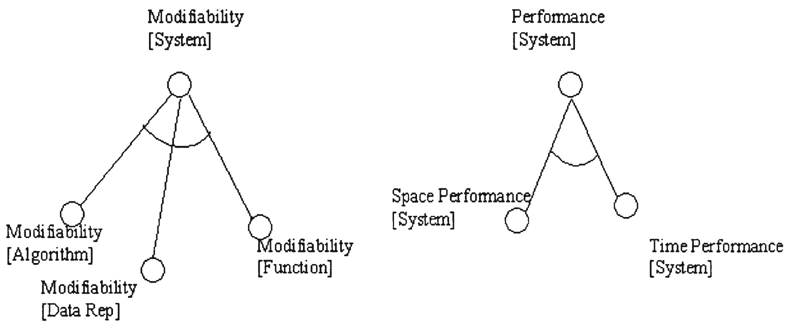


Figure 1. A softgoal interdependency graph showing refinements of quality requirements based on topic and type

In parallel with the refinement of quality requirements, the software architect will consider different ways of meeting the KWIC functional requirements in the context of the web information system. At various points during the design process, the architect will go through a number of interleaving activities of componentization, composition, choice of architectural style, etc. Each activity can involve consideration of alternatives, where NFRs can guide selection, hence narrowing down the set of architectural alternatives to be further considered.

For example, the architect can consider architectures with varying numbers of (main) components:

- Input, Circular Shift, Alphabetizer, and Output
- Input, Line Storage, Circular Shift, Alphabetizer, and Output
- etc.

Each choice will make particular contributions to the NFRs. With either choice the architect can further consider alternatives about control, for example, one with a Master Control and one without. Yet another decision point might concern the way data is shared: sharing of data in the main memory, sharing of data in a database, sharing of data in a repository with an event manager and so forth.

Figure 2 describes some of the above alternative architectures using "conventional" block diagrams. The diagrams were redrawn by one of the authors based on (Shaw & Garlan, 1996).

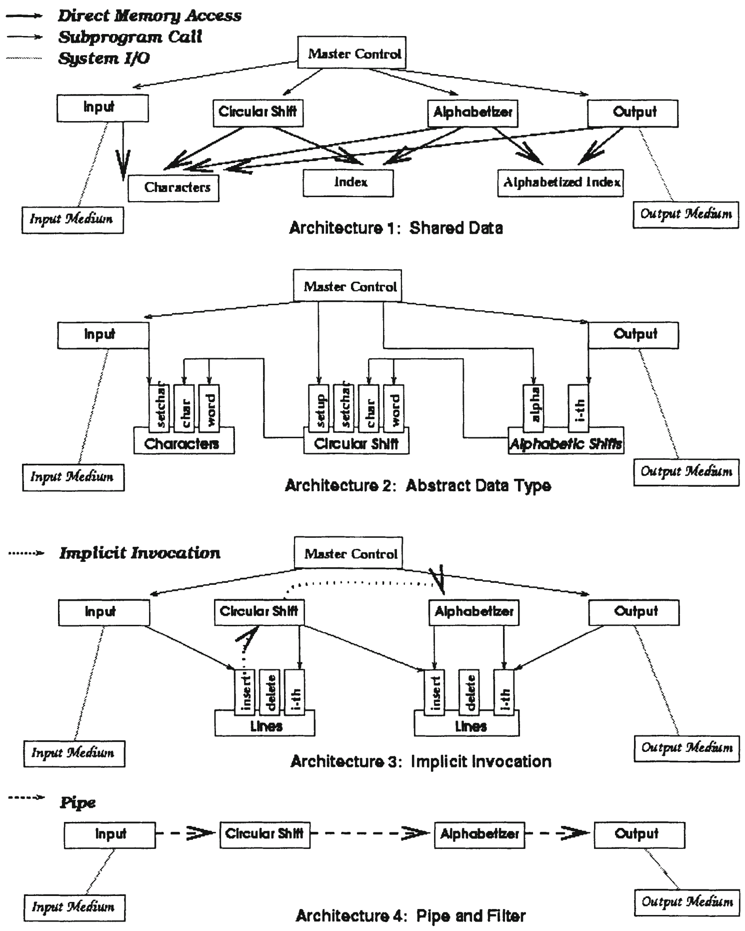


Figure 2. Architectural alternatives for a KWIC system

Let us assume that the architect is interested in an architecture that can contribute positively to the softgoal Modifiability [Data representation], and considers the use of an "Abstract Data Type" style of architecture, as discussed by Parnas (Parnas, 1972) , and Garlan and Shaw (Garlan, 1993): components communicate with each other by means of explicit invocation of procedures as defined by component interfaces.

As the architect would learn sooner or later, the positive contribution of the Abstract Data Type architecture towards modifiable data representation is made at the expense of another softgoal, namely the time performance softgoal. Figure 3 shows the positive contribution made by the abstract data type solution by means of "+" and the negative contribution by "-" *contribution link.*

The architect would want to consider other architectural alternatives in order to better satisfice the stated softgoals. The architect may discover from the literature that a "Shared Data" architecture typically would not degrade system response time, at least when compared to the Abstract Data Type architecture, and more importantly perhaps it is quite favorable with respect to space requirements. This discovery draws on work by Parnas (Parnas, 1972), and by Garlan and Shaw (Garlan, 1993) who considered a Shared Data architecture in which the basic components (modules) communicate with each other by means of shared storage. Not unlike the Abstract Data Type architecture, however, the Shared Data architecture also has some negative influence on several other softgoals: a negative (-) impact on modifiability of the underlying algorithm (process) and a very negative (--) impact on modifiability of data representation.

Figure 3 shows both design steps along with the various contributions that each alternative makes towards the refined softgoals. Note that the diagram is build iteratively rather than in one step, according to the architectural "discovery process" of the architect.

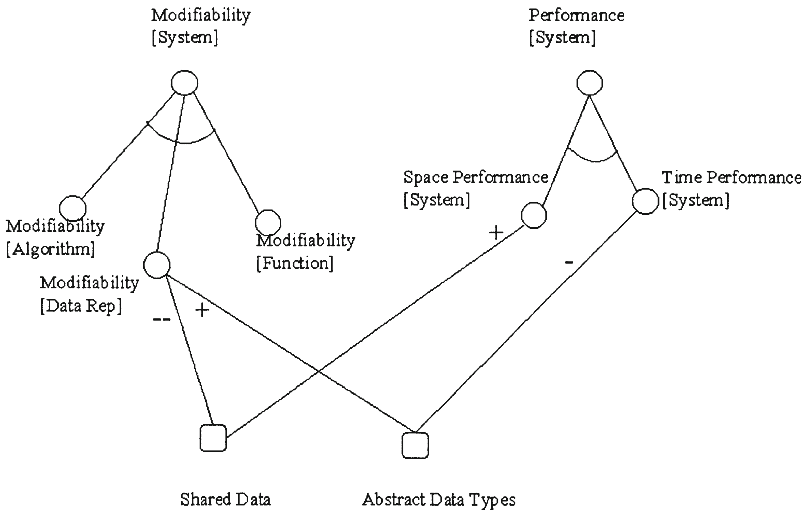


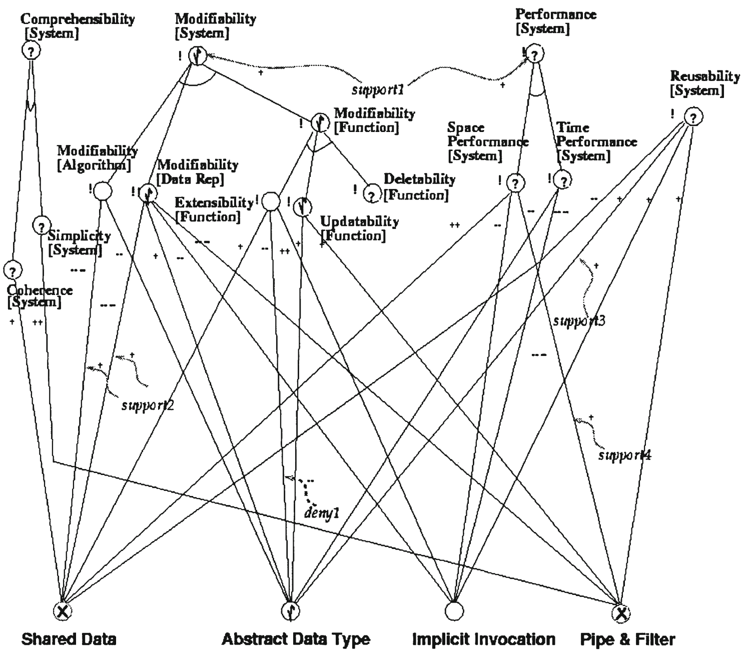
Figure 3. Contribution of the Shared Data and Abstract Data Type architectures

Interestingly, Figure 3 shows tradeoffs between the architectural alternatives that have been considered so far. The architect can continue to consider other architectural alternatives, including hybrid solutions, or decide which of the two better suits the needs of the stakeholders. How can the architect go about doing the latter, if that is what she so desires? One way to do the tradeoff analysis is by using the degree of *criticality* (or priority, or dominance, or importance) of the quality requirements. In the context of a particular web information system, for example, the stakeholders might indicate that performance is more critical than modifiability. In this case, then, the architect would choose Shared Data over Abstract Data Type, since Shared Data is more satisfactory with respect to both space and time performance, hence the overall performance requirements (recall the "AND refinement").

During the process of architecting, the architect needs to make many decisions, most likely in consultation with stakeholders. As the above discussion suggests, an interesting question is: "how can the architect evaluate the impact of the various decisions?" The NFR Framework provides an interactive evaluation procedure, which propagates labels associated with softgoals representing their satisficing status (such as *satisficed, denied, undetermined, and conflict*) across the softgoal interdependency graph. Labels are propagated along the direction of contribution, usually "upwards" from specific, refined goals towards high level initial goals.

Because of the subjective nature of quality requirements, the software architect will want to explain and justify her decisions throughout the softgoal refinement process. This can be done in the NFR Framework using "*claims*". Claims can be attached to contributions (links in the graph) and to softgoals (nodes). Claims can themselves be justified by further claims. These rationales are important for facilitating understanding and evolution. For example, Shared Data may by and large have advantage over Abstract Data Type with respect to space consumption. This general relationship, however, may need to be argued for (or against), in the context of the particular web information system. If, for example, the volume of the data to be maintained by the system is low, the relative advantage of Shared Data may not matter much. If this is indeed the case, the expected data volume can then be used as a claim against the relationship: "Shared Data makes a strong positive (++) contribution towards meeting space requirements". This might then lead the architect to choose Abstract Data Type as the ultimate architecture.

Figure 4 shows a softgoal interdependency graph for the KWIC system, taken from work by Chung, Nixon and Yu (Chung, 1995) which is based on (Garlan, 1993) and Garlan, Kaiser, and Notkin (Garlan, 1992).



support1: among the vital few goals (from market survey)

support2: [Parnas72]

support3: fewer asumptions among interacting modules

support4: expected size of data is huge (from domain expert)

deny1: many implementors familiar with ADTs (from domain expert)

Legend

Link Types		Criticality	Evaluation Labels	
+	weak positive satisficing		?	undetermined
++	strong positive satisficing	! critical	○	neutral
--	weak negative satisficing	!! very critical	✓	satisficed
---	strong negative satisficing		✗	denied

Figure 4. A softgoal interdependency graph for the KWIC system

3. MEETING DIFFERENT STAKEHOLDER REQUIREMENTS

We now illustrate the need to relate organizational context to the process, and consequently the outcomes, of architectural design. The illustration will be done through three scenarios, which will show that different sets of stakeholder concerns are transformed by the architectural design process into different architectural choices for information systems. More specifically, each different set of stakeholders and their concerns leads the architects to reason about different quality concerns, make and evaluate different design decisions, and finally leads, in our case, to the most appropriate architectural designs to be used in a particular web-based information system context.

3.1 Scenario 1

An *e-shopping software vendor* specializes in offering software products, which can be used in advertising, selling, and shipping goods and services in an Internet-based virtual market. The products should generate, among other things, e-catalogs so that any Internet user can search for goods using a web-browser. The *e-catalog application architect* realizes that she needs a component which can generate an index, here an alphabetized list of the words in the descriptive text of each catalog item such that each word in the list is associated with a list of all catalog items pertaining to that word. Such a list, however, is just what a KWIC system generates. Hence, the *e-catalog application architect* asks a *KWIC component architect* to built an indexing system. This is a brief description of the essential functional aspect of the scenario. We will shortly describe the quality aspect of the scenario, along with more details of the functional aspect.

Figure 5 depicts the relationships among the three types of stakeholders, using the *i** framework proposed by Yu (Yu, 1994). The *i** framework allows for the description of *actors* and their *dependencies* in organizational settings. A circle represents an actor (e.g., *e-shopping software vendor*) who may be dependent on some other actor (e.g., *e-catalog application architect*) to achieve some of its goals (e.g., developing an e-catalog application). Not unlike the NFR Framework, the *i** framework also distinguishes a quality requirement, denoted by a cloud like shape (to suggest softness), from a functional one, denoted by a rectangle. In the *i** framework , a dependency is described by a directed link between two actors. The semi-circle on the directed link stands for the letter "D" which denotes the notion of dependency. This type of graph is called a Strategic Dependency model in the *i** framework (the other type of graph in *i**; the Strategic Rationale model will not be discussed in this paper).

In the current scenario, the *e-shopping software vendor* depends on the *e-catalog application architect* to deliver an e-catalog application, who in turn depends on the *KWIC component architect* to deliver an indexing system.

This kind of diagram shows where requirements originate. It also serves as a basis for determining what kind of negotiated delegations should take place, how different architectural decisions affect the various stakeholders, and possibly what kind of requirements to allocate to, and how to partition the system into, sub-systems and components. Just like a softgoal interdependency graph, it becomes a basis for justification and system/software architectural evolution.

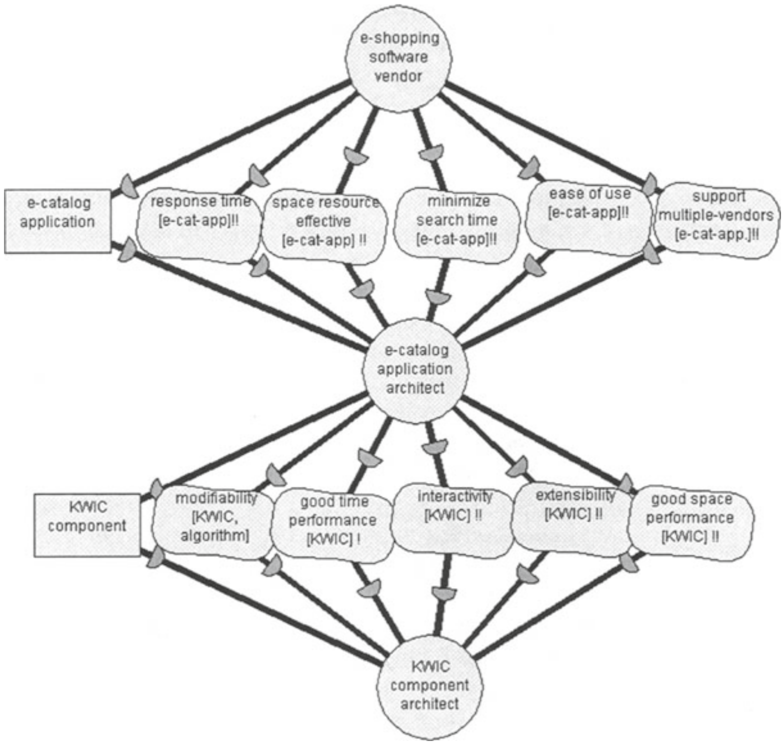

e-shopping software vendor
e-catalog application
response time [e-cat-app]!!
space resource effective [e-cat-app] !!
minimize search time [e-cat-app]!!
ease of use [e-cat-app]!!
support multiple-vendors [e-cat-app.]!!
e-catalog application architect
KWIC component
modifiability [KWIC, algorithm]
good time performance [KWIC] !
interactivity [KWIC] !!
extensibility [KWIC] !!
good space performance [KWIC] !!
KWIC component architect

Figure 5. Organizational context for the e-catalog application

Now we describe the quality concerns of the stakeholders. To start with, the e-shopping software vendor expects the application to be easy to use. The vendor also has other concerns. As the catalog items are expected to grow quite rapidly, storage space resource is a very important concern, as is fast response time. Also shown in Figure 5 is multiple-vendor support, namely, allowing for the integration of catalogs that reside on various server machines in physically remote vendor organizations. The exclamation marks denote the criticality of a quality. The highest priority is assigned to two exclamation marks, medium priority to one, and low priority none.

As a matter of fact, the list of quality requirements and their criticalities is determined through cooperation between the e-catalog application architect and the e-shopping software vendor who go through a process of recursive refinements, in the manner of the previous section, which may also require the KWIC component architect's involvement at least occasionally. The list then becomes what is commonly known as the user requirements.

When the user requirements are more or less satisfactory, the e-catalog application architect directs her attention more towards defining the system requirements, whose clarification may need more of the KWIC component architect's involvement than before. The system requirements may inherit some of the user requirements more or less directly, such as good space and response time requirements. The system requirements will also come from the system's perspective. For example, the "ease of use" requirement now may be translated more specifically into interactivity (such as configuring indexing options dynamically) and extensibility (such as allowing for the use of international language character sets, categorical search and phonetic search). Another system requirements that might be considered is the modifiability requirement, here for changing the overall algorithm which builds those indices transparently in a distributed setting. The criticalities may also change, due to the new requirements and the derived requirements. For example, in the presence of the extensibility requirement, which is new, the criticality of the good time performance requirement is lowered from critical to medium.

With the organizational context in place, the KWIC component architect uses the process-oriented NFR Framework to refine the quality softgoals, consider architectural design alternatives, carry out tradeoff analysis and evaluate the degree to which softgoals are satisficed, all in consideration of the context. The top portion of Figure 6 represents those softgoals that originated from the e-shopping software vendor, and are negotiated and delegated through the e-catalog application architect to the KWIC component architect. The relative criticality values are preserved in the softgoal interdependency graph. Figure 6 shows the result of the process whereby the architect has arrived at four architectural alternatives in an attempt to satisfice the stated softgoals.

Importantly, the diagram in figure 6 shows a number of claims, which derive from the knowledge of the organizational context, and which are used to argue for, or against, the types of softgoal criticalities and interdependencies, and consequently in softgoal evaluation and selection among architectural alternatives. For example, using the Shared Data architectural style is expected to have a very good contribution towards space performance. The architect uses the organizational context diagram (figure 5) to find some argument in support (or denial) of that particular contribution. In the current scenario, for example, the architect argues for the validity of the contribution by pointing to the e-shopping software vendor who wants the system to have the ability to handle a rapidly growing number of catalog items. The “S1” arrow in figure 6 denotes this claim.

Despite the significant savings by the Shared Data architecture in data storage, however, the Implicit Invocation architecture seems to be the most promising for achieving extensibility of function, which is as critical as space performance. Furthermore, Implicit Invocation helps modifiability of the algorithm, in contrast to Shared Data, although there is a tie between the two concerning interactivity. Although not well met by Implicit Invocation, time performance is of lower criticality. Taking all these into account, the KWIC component architect chooses the Implicit Invocation as the target architectural design.

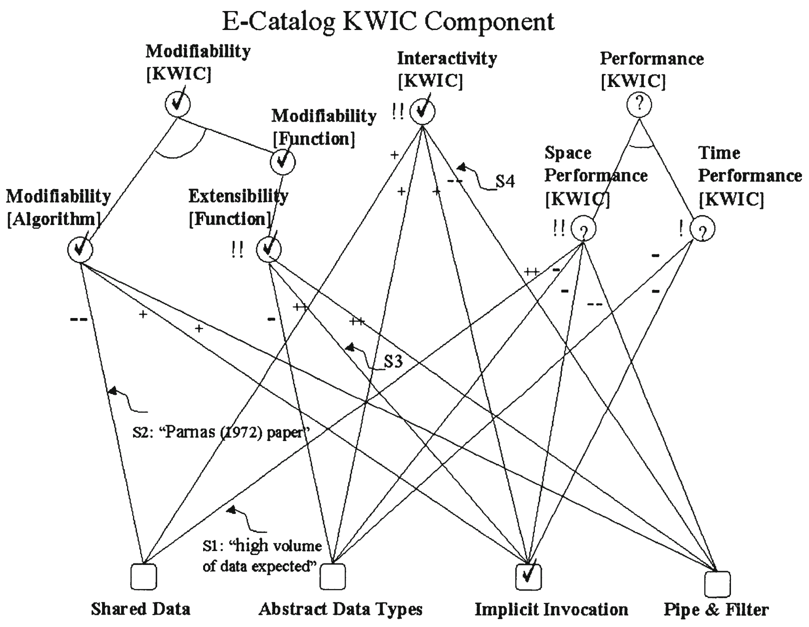


Figure 6. A softgoal interdependency graph for the e-catalog KWIC component

3.2 Scenario 2

A system administrator wants to offer the user a help facility, which can retrieve all the documents that have some keyword in their description, as indicated by the user. The administrator asks a help system architect to build such a facility. The help system architect, in turn, asks a KWIC component architect for an indexing software system, after realizing that the facility is essentially a KWIC system such as used in the Unix “man –k” command.

Similar to figure 5 for scenario 1, we may now describe the three types of stakeholders using the *i** framework, together with the functional and quality requirements that the stakeholders delegate among themselves, together with the various criticalities of each of the requirements. And analogous, to figure 6 for scenario 1, the architect iteratively builds an NFR softgoal interdependency graph in which she further refines the various quality requirements and argues for or against certain claims. These analogous figures for scenario 2 are not shown for lack of space, but some fragments of the functional and quality requirements as well as the (soft) goal interdependencies related to this scenario appear in figure 7 and 8.

Taking all contributions of each architectural style into account, together with the various criticalities of the softgoals to be achieved, the architect might want to choose the Pipe and Filter architectural style as the most promising one.

3.3 Scenario 3

A reuse manager is appointed by product line management to oversee the development of various systems in the organization. As it happens, the KWIC component architect, the e-catalog application architect and the help system architect all work in the same organization. The reuse manager asks the architects to consider reuse as a critical priority and to maximize reuse of all components developed in that organization.

This scenario is especially interesting as it introduces a stakeholder (the reuse manager) whose quality concern (having reusable components) prompts the KWIC component architect to find a solution that represents the union of quality concerns of all other architects, as well as taking into account each of their intended customer (the e-shopping software vendor and the man administrator).

Essentially, figure 7 shows a merge of all stakeholders' quality softgoals discussed in the previous scenarios. In addition it show that reuse manager depends on the KWIC system architect to build a system that delivers and maximizes the use of reusable components for all development activities in that organization. Not shown are product line management stakeholders, who depend on the reuse manager for reduced development costs.

For each of the two previous scenarios a different architectural solution style was chosen according to the specific kind of organizational context and its derived set of requirements. To find a reusable component solution the KWIC component architect will need to re-negotiate the delegated requirements with each of the involved stakeholders to overcome the stakeholders conflicting requirements. Perhaps the KWIC component architect will also need to renegotiate the degree of reusability with the reuse manager.

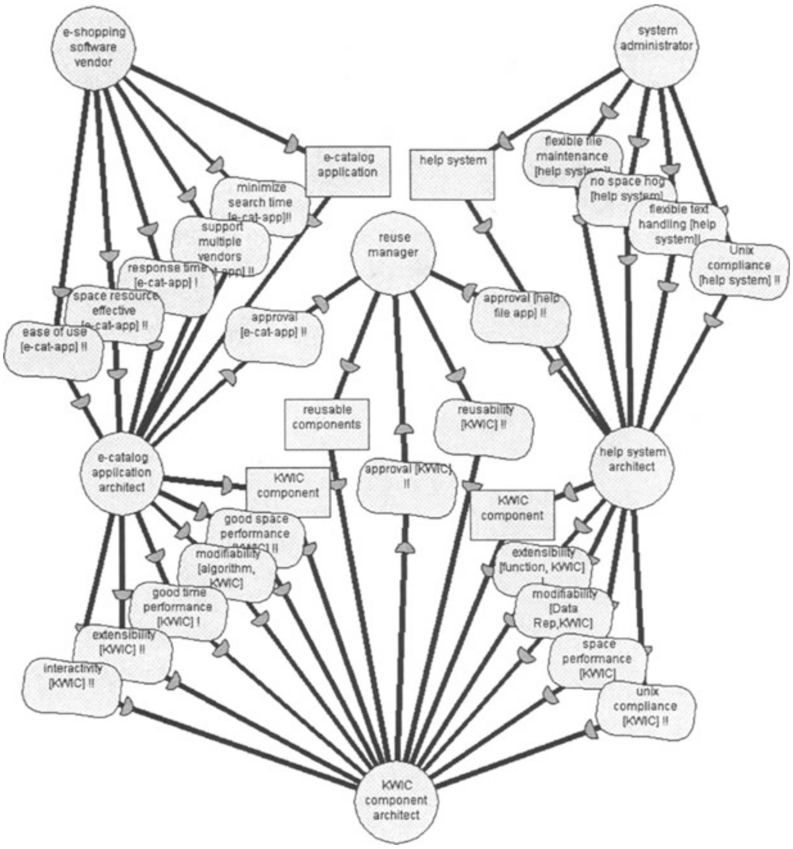

e-shopping software vendor
system administrator
e-catalog application
help system
minimize search time [e-cat-app]!!
support multiple vendors
response time [e-cat-app] !
space resource effective [e-cat-app] !!
ease of use [e-cat-app] !!
approval [e-cat-app] !!
reuse manager
flexible file maintenance [help system]
no space hog [help system]
flexible text handling [help system]
Unix compliance [help system] !!
approval [help file app] !!
reusable components
reusability [KWIC] !!
approval [KWIC] !!
e-catalog application architect
help system architect
KWIC component
good space performance [KWIC] !!
modifiability [algorithm, KWIC]
good time performance [KWIC] !
extensibility [KWIC] !!
interactivity [KWIC] !!
extensibility [function, KWIC]
modifiability [Data Rep,KWIC]
space performance [KWIC]
unix compliance [KWIC] !!
KWIC component architect

Figure 7. Organizational context for the reuse requirement

Now that the KWIC component architect has an organizational understanding (of which quality requirements and criticalities originated from which stakeholders, and what network of relationships exists among them), she proceeds to use the NFR framework to evaluate, and further argue for or against the various architectural styles. During the evaluation, the architect renegotiates conflicting quality requirements and criticalities with the affected stakeholders and finds an architectural solution that makes acceptable tradeoffs. Figure 8 shows the result of the architectural design process. (The "broken" lines are not part of the NFR Framework graphical notation, but are used in this paper to avoid cluttering the diagram with links not directly related to the architectural styles shown to be evaluated. The "e" subscript stands for the e-catalog application architect's point of view, the "h" subscript stands for help system architect's point of view, while the "r" subscript stands for the reuse mangers point of view).

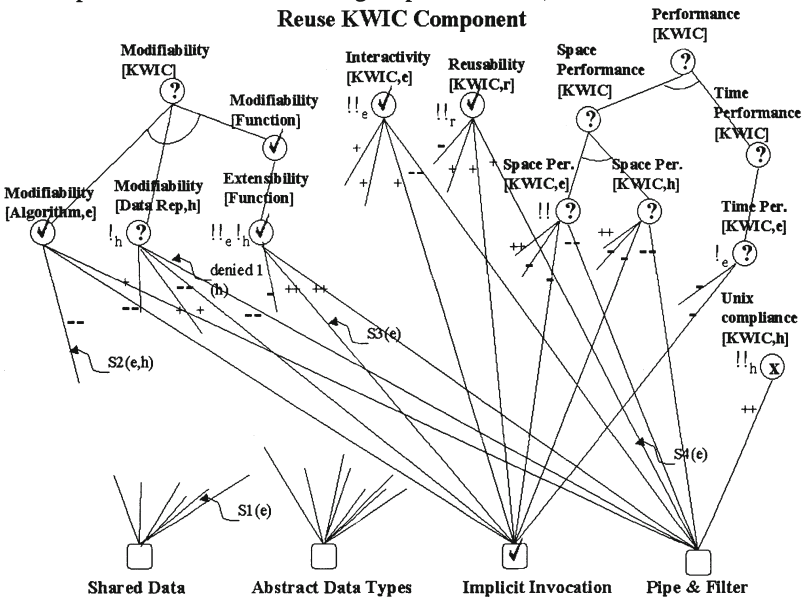


Figure 8. A softgoal interdependency graph for the reusable KWIC component

The figure shows the KWIC component architect evaluating the Implicit Invocation style for meeting the quality requirements originating from the e-catalog application architect, the help system architect and the reuse managers. While evaluating the Implicit Invocation style the architect may renegotiate with the help system architect her demand for "Unix compliance" which, for her, would be better dealt with when using the Pipe & Filter style. The organizational context (such as the "approval" dependency that the architects have on the reuse manager), will make the negotiating parties more forthcoming when concessions to their requirements and/or criticalities are needed.

4. DISCUSSION AND RELATED WORK

As pointed out by Garlan and Perry (Garlan, 1994), architectural design has traditionally been largely informal and ad hoc. Our proposal is aimed at rectifying some of the manifested symptoms by taking a more disciplined approach to architectural design. In particular, our proposal is aimed at improving our ability to understand the rationales behind architectural choices, hence making the system more easily traceable and evolvable. We have illustrated how to carry out a finer-grained analysis, and the comparison of architectural designs by considering quality-related concerns of multiple stakeholders and their interdependencies.

Our proposal draws on concepts that have been identified as essential to portray architectural infrastructure, such as elements, components, and connectors as suggested by Perry and Wolf (Perry, 1992), Garlan and Shaw (Garlan, 1993), Abowd, Allen, and Garlan (Abowd, 1993), and Robbins, Medvidovic, Redmiles and Rosenblum (Robbins, 1998). In our view, our emphasis on quality concerns and stakeholder interdependencies are complementary to efforts directed towards identification and formalization of concepts for functional architectural design.

Concerning the role of quality requirements, design rationale, and assessment of alternatives, the proposal by Perry and Wolf (Perry, 1992) is of close relevance to our work. Perry and Wolf propose to use architectural style for constraining the architecture and coordinating cooperating software architects. They also propose that rationale, together with elements and form, constitute the model of software architecture. In our approach, weighted properties of the architectural form are justified with respect to their positive and negative contributions to the stated NFRs, and weighted relationships of the architectural form are abstracted into contribution types and labels, which can be interactively and semi-automatically determined.

Boehm (Boehm, 1992), and Kazman, Bass, Abowd, and Webb (Kazman, 1994) have argued convincingly for the importance of addressing quality concerns in software architectures. Kazman, Bass, Abowd, and Webb (Kazman, 1994) propose a basis (called SAAM) for understanding and evaluating software architectures, and gives an illustration using modifiability. This proposal is similar to ours, in spirit, as both take a qualitative approach, instead of a metrics approach, but differs from ours since SAAM is product-oriented, i.e., they use quality requirements to understand and/or evaluate architectural products.

In comparing architectural alternatives, it is intuitively appealing to use a tabular format. For example, in (Garlan & Shaw, 1993), a table is used to present the quality evaluations of four architectural alternatives. Such a table can be interpreted as depicting contributions from the architectural alternatives to the quality attributes treated as goals. In our study, we illustrated the importance of context and the need to trace design decisions to stakeholder requirements. Our approach suggests that the tabular representation of design alternatives and quality attributes is not sufficiently expressive.

We might consider extending the tabular representation by distinguishing quality requirements that come from different stakeholders, and by adding more explanatory notes such as the claims in the softgoal interdependency graphs.

Our approach emphasizes explicitly representing and using the quality concerns of multiple interacting stakeholders during the design of software architectures. Our approach is thus similar to the on-going work by Boehm and In (Boehm, 1996), who explore a knowledge-based tool for identifying potential conflicts among quality concerns early in the software/system life cycle, and using quality requirements in examining tradeoffs involved in software architectural design. Stakeholders such as user, maintainer, developer, customer, etc., are mapped to quality attributes in a graph. Our approach goes further by indicating that stakeholder requirements can be traced through a network of dependency relationships in an organizational model.

5. CONCLUSIONS AND FUTURE WORK

Achieving architectural quality requirements is a key objective in architecture-based approaches to software engineering. Quality requirements vary according to context and need to be negotiated among stakeholders. We have outlined a systematic approach for representing and addressing quality requirements during architectural design. The design reasoning is related to context through an organization model of stakeholder dependencies.

Using an extended version of the familiar KWIC example, we have illustrated how architectural decisions might vary depending on context, and how the design process can be guided and assisted using appropriate notational and reasoning support. The historical records of design decisions and rationales will facilitate understanding and evolution.

We have been working on tools to support the approach. These include facilities for generating and maintaining the graphs, for propagating labels, and for design revision. Knowledge for addressing specific quality requirements are codified in knowledge bases to assist in the refinement of goals. Known interactions among quality requirements are codified as correlation rules for detecting conflicts and synergies.

This paper represents a first step in an attempt to provide a systematic architectural design support framework that takes organizational and stakeholder relationships into account. We have drawn on the NFR framework for dealing with software quality requirements, and the *i** framework for modeling and reasoning about strategic actor relationships. In future work, we intend to further elaborate on issues specific to architectural design, and to better integrate architectural design reasoning and organizational relationships reasoning.

REFERENCES

Abowd, G., Allen R. and Garlan, D.(1993) "Using Style to Understand Descriptions of Software Architectures", *Software Engineering Notes*, 18(5): 9-20, *Proc. of SIGSOFT `93: Symposium on the Foundations of Software Engineering.*

Boehm, B. W. (1976) "Software Engineering", *IEEE Transactions on Computers,* 25(12), pp. 1226-1241

Bass, L., Clements P. and Kazman, R. (1998) Software Architecture in Practice, *SEI Series in Software Engineering*, Addison-Wesley.

Boehm, B. and Scherlis, B(1992) "Megaprogramming", *Proc. the DARPA Software Technology Conference.*

Boehm, B. and In, H.(1996) "Aids for Identifying Conflicts Among Quality Requirements", *Proc. International Conference on Requirements Engineering*, (ICRE96), Colorado, April 1996, and *IEEE Software*, March 1996.

Bowen, T. P. , Wigle, G. B. and Tsai, J. T. (1985) "Specification of Software Quality Attributes", Report RADC-TR-85-37, vol. I (Introduction), vol. II (Software Quality Specification Guidebook), vol III (Software Quality Evaluation Guidebook), Rome Air Development Center, Griffiss Air Force Base, NY, Feb. 1985.

Chung, L.K.(1993) "Representing and Using Non-Functional Requirements: A Process-Oriented Approach*". Ph.D. Thesis, Dept. of Computer Science, Univ. of Toronto*, June 1993. Also Technical Report DKBS-TR-93-1.

Chung, L.K. Nixon, B. and Yu, E.(1995) "Using Non-Functional Requirements to Systematically Select Among Alternatives in Architectural Design", *Proc., 1st Int. Workshop on Architectures for Software Systems*, Seattle, April 24-28, 1995., pp. 31-43.

Chung, L.K. Nixon, B. A., Yu, E and J. Mylopoulos(1998), Non-Functional Requirements in Software Engineering, Kluwer Publishing (to appear).

Garlan D. and Shaw, M.(1993) "An Introduction to Software Architecture *Advances in Software Engineering and Knowledge Engineering: Vol. I,* World Scientific Publishing Co.

Garlan, D., Kaiser, G. E. and Notkin, D. (1992) "Using Tool Abstraction to Compose Systems", *IEEE Computer,* Vol. 25, June 1992. pp. 30-38.

Garlan, D. and Shaw, M. (1993) "An Introduction to Software Architecture", *in Advances in Software Engineering and Knowledge Engineering: Vol. I,* World Scientific Publishing Co.

Garlan, D. and Perry, D.(1994) "Software Architecture: Practice, Potential, and Pitfalls", *Proc. 16th Int. Conf. on Software Engineering*, pp. 363-364.

Kazman, R, Bass, L., Abowd, G. and Webb, M. (1994) "SAAM: A Method for Analyzing the Properties of Software Architectures", *Proc. Int. Conf. on Software Engineering*, May 1994, pp. 81-90.

Nixon, B. A.(1993) "Dealing with Performance Requirements During the Development of Information Systems.", *Proc. IEEE Int. Symp. on Requirements Engineering*, San Diego, CA, January 4-6, Los Alamitos, CA: IEEE Computer Society Press, pp. 42-49.

Parnas, D. L. (1972) "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, Vol. 15, Dec. 1972, pp. 1053-1058.

Perry, D. E. and Wolf, A. L. (1992) "Foundations for the Study of Software Architecture", *ACM SIGSOFT Software Engineering Notes*, 17(4), pp. 40-52.

Robbins, J. E. , Medvidovic, N., Redmiles, D. F. and Rosenblum, D. S. (1998) "Integrating Architecture Description Languages with a Standard Design Method*", Proc. 20th Int. Conf. on Software Engineering*, pp. 209-218.

Shaw, M. and Garlan, D. (1996) "Software Architecture: Perspectives on an Emerging Discipline", Prentice Hall.

Yu, E. S. K. and Mylopoulos, J. (1994) "Understanding `"Why"' in Software Process Modelling, Analysis, and Design.", *Proc., 16th Int. Conf. on Software Engineering*, Sorrento, Italy, May 1994, pp. 159-168.

Yu, E.(1995) "Modelling Strategic Relationships for Process Reengineering", *Ph.D. Thesis, Dept. of Computer Science, Univ. of Toronto.*

APPENDIX

The KWIC problem statement (Parnas, 1972): "The KWIC [Key Word in Context] index system accepts an ordered set of lines; each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a list of all circular shifts of all lines in alphabetical order."

The Software Architect
—and the Software Architecture Team

Philippe Kruchten
Rational Software, 650 West 41st Avenue # 638, Vancouver, B.C., V5Z 2M9 Canada
pbk@rational.com

Key words: Architecture, architect, team, process

Abstract: Much has been written recently about software architecture, how to represent it, and where design fits in the software development process. In this article I will focus on the people who drive this effort: the architect or a team of architects—the software architecture team. Who are they, what special skills do they have, how do they organise themselves, and where do they fit in the project or the organisation?

1. AN ARCHITECT OR AN ARCHITECTURE TEAM

In his wonderful book *The Mythical Man-Month*, Fred Brooks wrote that a challenging project must have one architect and only one. But more recently, he agreed that "Conceptual integrity is the vital property of a software product. It can be achieved by a single builder, or a pair. But that is too slow for big products, which are built by *teams*."[1] Others concur: "The greatest architectures are the product of a single mind or, at least, of a very small, carefully structured team."[2] More precisely: "Every project should have exactly one identifiable architect, although for larger projects, the principal architect should be backed up by an architecture team of modest size."[3]

[1] Keynote address, ICSE-17, Seattle, April 1995
[2] Rechtin 1991, p. 22
[3] Booch 1996, p. 196

We speak about a *software architecture team*, and assume that the lone architect is just a simpler case. We speak of a *team*, not just a working group or a committee; a team in the sense defined by Katzenbach and Smith in *The Wisdom of Teams*: "a small number of people with complementary skills who are committed to a common purpose, performance goals, and approach for which they hold themselves mutually accountable."[4]

2. SKILLS OF THE ARCHITECTS

Software architects must collectively have a certain number of skills: experience (both in software development and in the application domain), good communication skills, sense of leadership, they are proactive, and goal-oriented.

2.1 A broad range of experience

Software architects must have accumulated significant experience in *software development*, especially if they are to tackle ambitious projects, but at the same time, they must be (or should become) knowledgeable in the *problem domain*. The two kinds of expertise must be well balanced. Ambitious software architecture projects will not succeed without *both*.

If the architects have a good understanding of the problem domain, such as telephony, air-traffic control, or computer-aided manufacturing, but only limited experience with software development and software architecture, they will not be able to rapidly develop an architecture that can be communicated to the various development groups. Even if they do not develop the code themselves, they must master the software design method (e.g., OOD), the programming language(s), understand the development environment, the development process, because they will have to interact daily with the software designers, programmers, and database engineers. They have to understand them and be understood. Their design decisions must be acceptable by software engineers. They must make some decisions very quickly, based on experience and "gut feelings" rather than pure, thorough analysis.

In one case the resentment against an architecture team was growing. When we were called to help, we discovered that they were excellent people with a lot of experience in their field, doing a very good job of analysis, building a very solid, object-oriented model of their domain, but carefully avoiding making any software design decisions. All questions

[4] Katzenbach 1994, p. 45

> about the "how" were brushed aside as "mere implementation details" that should not pollute their architectural description. Further investigation showed that they were in fact afraid of making any technical choices because of their lack of experience with similar systems. They were also under psychological pressure from technical leaders of the various development teams who they thought were much more qualified to speak about the software itself. Therefore they had shifted their focus toward analysis, even though the rest of the software development organisation was still holding them accountable for the major architectural decisions.

When architects have a good grasp of the software development aspect but a poor knowledge of the domain, they will develop nice solutions for the wrong problems, reduce the real problems to problems they know how to solve, or impose solutions that suit software engineers but are for a user community that works, behaves and thinks completely differently. For example, air-traffic controllers are not software developers; they have another view of the usefulness of menus and windows rather than the views shared by most software engineers. Imposing Macintosh-like desktop metaphors because it proved to be good for software types may prove to be a mistake in this specific case.

If you agree that software architecture, like building architecture, is concerned with more than the nuts and bolts of the software, such as how the software is used in its context—sociological and economical i.e., looking outwards, and not merely inwards—then it becomes clearer why a software architecture team must be versed in both software development and the application domain. Architects need to anticipate changes, changes in the environment in which the system under development will be deployed, which will in turn trigger requests for change or evolution. You can only do this if you are looking at that context, that domain, not just looking at the software itself. Architects need to develop a long-term vision for the project: where do we want to be with this software in two years, five years, and ten years from now?

Software architects are curious, they keep their ears and eyes open, read technical publications, and try to constantly sharpen their skills, extending and broadening the scope of their knowledge. They develop their creative skills by looking at other fields, other domains, other disciplines, from which they can derive more analogies.

Achieving this balance of expertise in a software architecture team is hard. It is not enough to bring together a few people that are very good at software development and a few people that are good at the problem domain; they must have enough knowledge, language, and vision in common so they can communicate and produce something.

In another case, when we were creating a team to develop an architecture for an air-traffic control (ATC) system, we identified some talented software designers, and some talented ATC specialists. This was just the beginning. All the software people were sent for hands-on training on air-traffic control, going to ATC classes, then spending days sitting next to controllers in a live Area Control Centre, trying to understand the essence of their activity. Similarly, the ATC specialists were sent to courses such as Object-Oriented Design, Programming in Ada, to reach the point where there was enough common vocabulary for them to efficiently work together and leverage each other's skills.

This approach does not always work without resistance: "Why should I learn about programming, since I will never program?", "Why should I waste my time going through air-traffic controller training? I am a software designer..."

The real difficulty is when there is only one architect: that one person must therefore be knowledgeable in both software development and the domain. Finding such people on the market is not very easy. The few we know of who are like that developed their unique combination of skills in a given organisation or company.

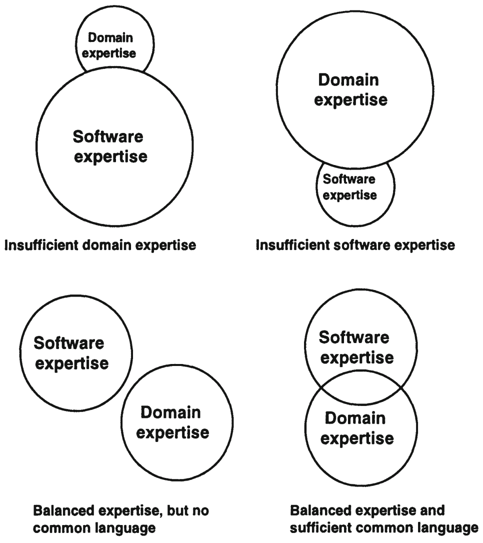

Domain expertise
Software expertise
Insufficient domain expertise
Domain expertise
Software expertise
Insufficient software expertise
Software expertise
Domain expertise
Balanced expertise, but no common language
Software expertise
Domain expertise
Balanced expertise and sufficient common language

Figure 1. Looking for the balanced team

When the scope of the project is large, the problem of finding people with balanced expertise and a common language gets worse. Within the software development domain, you may need to gather enough expertise from various "specialities" such as data engineering, operating systems, networking and security expertise. Within the application domain, there may be also specialisation: in telecommunications, for example, there is voice communication, call handling, voice messaging, versus data communication and packet switching. It is not possible to find people that are experts in all specialities, but collectively, the architecture team must be reasonably aware of these specialities so they can bring in and interact with experts whenever necessary.

The issue of a common language is important. By language, we mean as well as a common spoken language, they must have a common way of representing the architecture, and a common programming language. The choice of a language is a choice of a *model*, complete with its opportunities for creativity, its internal assumptions, and its constraints. Languages, among people who speak them, provide a rapid transfer of knowledge, imply consent with the underlying connectives, and agreement on stated conclusions.[5]

The *wider* the experience the better. People who have been working with the same kind of architecture for 20 years have 20 years of experience but are likely to reproduce that same architecture for a new project. A person with only 12 years of experience with three or four different kinds of architecture brings more experience to the table. Consider getting help from external consultants: the very nature of their job ensures a wealth of experience, perhaps including work with your direct competitors.

In all cases, software architects must understand software sufficiently well to be reasonable programmers. An architect unable to express or sketch a concept in a programming language is as suspicious as a building architect who does not know how to use a T-square, a French curve, or a lettering pen; he is putting the project at risk by having a wider gap to bridge with other developers. Often I write code simply to understand what I design.

2.2 Communication skills

Communication issues grow exponentially with the size of the development organisation. Successful architects or architecture teams rapidly become a centre of technical communication in a project, and they spend a significant amount of their time interacting with one or more stakeholders: explaining the architecture to other software developers, to

[5] Rechtin 1991, p. 80

system engineers, to customers, users, prospects, marketing people, and managers. They must have therefore good written and verbal communication skills. Therefore, they must persuade, understand, dig out the real issues, convince the sceptical, and sell the architecture. As Jonathan Losk says: "Don't ever stop talking about the system."[6]

On two different, large command and control systems, we noticed that the lead software architect was dedicating more than *half of his time* just communicating what architecture was and why it was important.

Software architects must also *listen*, listen to the project worries, to recurring difficulties with certain tools, procedures, and design choices, while they are constantly adjusting, correcting or merely explaining software architecture, again and again. In many cases, they have to *negotiate*, finding compromises that can be accepted by several stakeholders.

Unfortunately, there isn't a strong correlation between good technical skills and good communication skills.

2.3 Leadership

Architects must have some leadership skills—technical leadership, that is. This technical leadership must be based on their knowledge and their achievement, not simply on some administrative decision. We do not mean that they are the project leaders or managers, but they will lead the software development in many ways: by establishing the structure in which all the software development will be hosted, by establishing the main design rules, by ensuring that the design principles are followed, by injecting new ideas, new solutions, and new techniques into the project to improve its productivity or the quality of the product, by coaching and mentoring newcomers or more junior people.

2.4 Proactive, goal-oriented and committed

A software architecture team is not a committee, meeting every so often to share ideas or discuss issues. It is not a review board, nor a think tank for top management composed of selected staff technologists. It consists of a small number of people fully committed to a very specific goal: designing an architecture. A committee which meets two hours a week cannot design an architecture. Software architects work at this *full time*. Only in some rare cases can a member of the team split his or her time between more than one activity. In particular, we think that the function of a software architect is

[6] Jonathan Losk, cited in Rechtin 1991, p. 292

rarely compatible with that of project manager, except for very small projects (eight people or less). We will address this point in another article.

Architects must be able to sustain a high degree of uncertainty and ambiguity. Their work often consists of a long succession of suboptimal choices, made in relative obscurity, i.e., without the luxury of examining all alternatives and all ramifications of the choices. Many people with scientific training— and this is aggravated by inexperience—cannot tolerate it for very long and will tend to defer the decision-making to others.

3. THE ROLE AND PURPOSE OF THE ARCHITECTURE TEAM

The architecture team share a common goal, or small set of goals. For the team to remain focused and efficient, the goals must be clearly defined, both to the architecture team and to their environment. This imposes the need to define (for a given project) what software architecture is, what are its boundaries, and in particular, what are the responsibilities and extent of authority of the architecture team, how decisions are being delegated, how to avoid "turf conflicts," and who is accountable.

One of the best ways to establish this, especially in environments where the concept of software architecture is new, is to create and publish a charter for the architecture team. Section 3.1 is a template we have successfully used on several projects, with small variations.[7]

3.1 The charter of a software architecture team

The software architecture team is responsible for evolving and maintaining the vision of the "*name your project*" software architecture.

The main activities of the software architecture team are:

- Defining the architecture of the software
- Maintaining the architectural integrity of the software
- Assessing technical risks related to the software design
- Proposing the order and contents of the successive iterations and assisting in their planning
- Consulting services to various design, implementation, and integration teams
- Assisting marketing in future product definition

[7] This text was pinned on the wall near my office at Hughes Aircraft of Canada during most of my time as the lead software architect for the Canadian Automated Air Traffic System.

The main deliverables from the software architecture team are documents: a software architecture document, some elements of software design documents, design and programming guidelines, iteration contents, meeting and review minutes and design audits of the running system.

3.1.1 Defining the architecture

The architecture of the software is the general framework in which all software design is performed. The architecture defines the major design elements, the way they are organised, structured, the way they interact, and the way they are to be used.

To ensure that the architecture will meet the needs of the various parts of the software and the external requirements, the architecture team works in close relationship with the various "domain" development teams, and the system architecture team. The architects' view is one of breadth, whereas the domain designers' is that of depth.

3.1.2 Maintaining architectural integrity

The architecture team is responsible for the development and maintenance of design and programming guidelines. The architecture team is involved in the organisation of design and code reviews to ensure that those guidelines are being followed, or to make them evolve as necessary. It plays a major role in the organisation of end-of-iteration "post-mortem" reviews.

All changes to major interfaces and all explicit violations of a design or programming rule must be approved by the architecture team. The architecture team is the final arbiter in matters of software aesthetics.

Finally, the software architecture team is involved in "change control board" decisions to resolve problems that have an impact on the software architecture or some critical interface.

3.1.3 Assessing technical risks

The architecture teams maintain a list of perceived technical software-related risks. The team may propose exploratory studies or prototypes to investigate the feasibility of a technical solution before inserting it in the architecture.

3.1.4 Proposing contents of iterations

The architecture team proposes the technical contents and the order of successive iterations by selecting a certain number of scenarios and a certain number of common mechanisms (services) to be studied and implemented. This technical proposal is completed and refined by the various development teams based on available personnel or customer requirements in terms of deliverables, availability of tools and COTS products, or needs of other projects. The architecture team then helps the various development teams with the transition from the architectural-level design to the more detailed design.

3.1.5 Consulting services

Because of their thorough understanding of the entire system, members of the software architecture team can provide assistance to various development teams as a floating resource for a given study, or, when needed, to help keep the project on schedule, or as "coaches" because of their specific skills or knowledge.

3.1.6 Product definition

In the context of a line-of-business of *Widgets*, the software architecture team provides some assistance in the definition of future products. It can help the marketing team with the prospective customer's requirement analysis, and during the study of the impact of a new product, shelter the development teams from too much disruption. Although it is not their main objective, the software architects play a major role in the project as facilitator or arbiter between the various product teams because of their various functions.

The software architecture team is accountable to the project manager. Its work is reviewed by the project technical staff, and a selection of senior software designers from the other product design teams. Project management can also evaluate the architects' contributions to the product by auditing their input in the final running system.

4. A TEAM AMONG OTHER TEAMS

Where do you hook an architecture team in your "organisation chart"? A software architecture team is a team of software designers and developers which should be organised no differently than any other software development team. It just happens that they are focused on different levels of abstraction or granularity, and may, on the average, be more experienced than others. But it would be a mistake to separate them, either in terms of the reporting structure or geographically, from the other software development groups they are supposed to interact with on a daily basis.

Being on the software architecture team is not a honorific position, nor a sinecure, it is not a staff position, nor a research job. Its schedule is tied to that of the other teams, which are its main customers. It reports to the same project manager.

One way to picture this is to consider the software architecture team as playing a symmetrical role to that of an integration and test team—the architecture team precedes the development teams, scouting the terrain, drafting the design, while the integration team follows, collecting debris and the wounded. In some circumstances we have called this the "engine, box cars, and caboose" model. The software architecture team is the engine pulling the train, the box-cars are the 'softcrafters' who are very good in one specific domain; and the integration team is the caboose, getting the pieces of software and making sure they can be integrated in a continuous manner. Note again that software architecture is not project management.

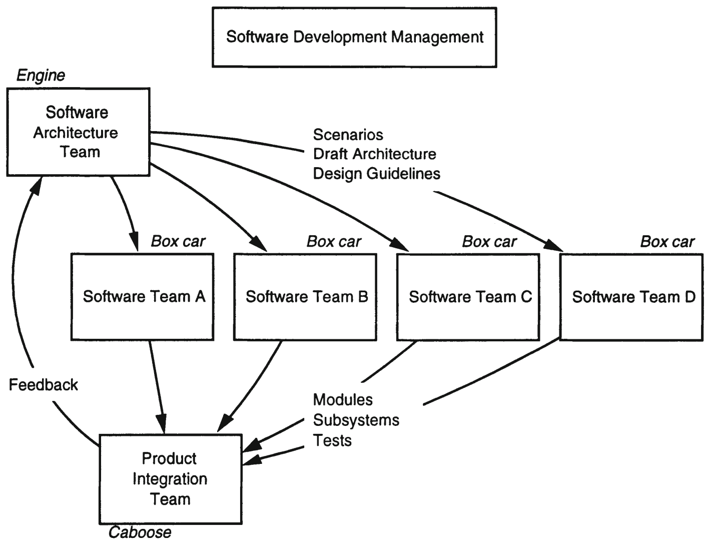


Figure 2. The engine and caboose model

This does not preclude a large company or organisation from having some R&D activity related to software architecture, nor some staff-level working group that overlooks the overall practice of software architecture across projects, but we are describing the software architecture team of a given project.

Some large organisations have set up two levels of software architecture activity:

- one group at the corporate level, whose purpose is to capture and diffuse the best practice in that area, or to oversee large-scale architectural reuse, and
- architecture teams closely associated to actual projects,

with some circulation of individuals from one group to another (cf. Fig. 3)

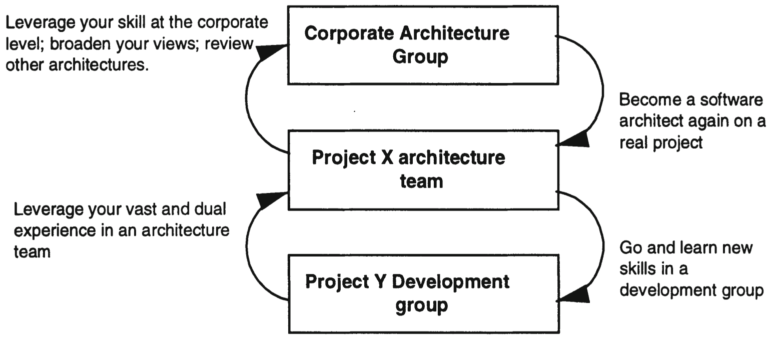

Leverage your skill at the corporate level; broaden your views; review other architectures.
Corporate Architecture Group
Become a software architect again on a real project
Project X architecture team
Leverage your vast and dual experience in an architecture team
Go and learn new skills in a development group
Project Y Development group

Figure 3. Recycling software architects

4.1 Size of the team

There is no simple absolute answer to this question. However, we will share a rule of thumb we have used which seems to correspond to the few data points we know from successful architecture teams that have been in place for years.

For a new, large, unprecedented project, one software developer out of 10 is on the architecture team during the inception and elaboration phase when architectural design is the preponderant activity. Then this can be reduced to one out of 12 or 15 as the project moves into the construction phase or during evolution cycles.

What happens to the disappearing architects? It is likely that the overall software development organisation will grow during the construction phase, hence the ratio is reduced. Also some of the software architects of the initial team can become technical leads for the various development groups. This evolution is beneficial from several aspects: since they have been part of the initial architecture team, they play a positive role in communicating the architecture and its principles to various parts of the project, or conversely, they bring new issues and difficulties to the attention of their former team-mates, with whom they had developed "high bandwidth communication links." In a large project, they play the role of "remote sensors" for the core architecture team, thereby contributing to maintaining the system's integrity. As a result, the evolution of the architecture through the latter phases of development can be smoother, based less on adversarial or conflicting relations between an architecture team and relatively foreign development teams.

Continuity is a key aspect. Eb Rechtin pleads for the architect to remain in charge until the project is delivered to the customer, whereas management may be tempted to "recycle" an architecture team to work on a new project as soon as the architecture of the current project is deemed "complete", i.e., at or soon after the end of the elaboration phase. The right balance is probably half way: keep enough architects on the project to guarantee the architectural integrity and to make any changes and improvements to drive it to a successful conclusion. This is where a team of architects offers more flexibility than a single architect.

4.2 System architecture and software architecture

In organisations that develop and integrate systems (composed of hardware and software, sometimes with all kinds of other devices), usually there is a strong *system* architecture function. Is software architecture just part of it? We have found that software issues are sufficiently distinct from system issues, and that the skill set of a software architect is significantly different from the other specialities present in the system architecture group to warrant the creation of a well-defined separate group to deal with software within the system architecture group.

However, software is more often the central issue, as hardware becomes more and more of a commodity, and the system architecture and software architecture functions tend to merge into a single entity.

5. TRAPS AND PITFALLS

Even when all of the pre-conditions are met, software architecture teams still fail. Over a large range of projects, Rational consultants have seen and analysed some of the reasons for these failures. We addressed some of them indirectly already such as

- Inexperience.
- Lack of domain experience
- Lack of software development experience
- Architecture team acts as a committee

Other reasons why the architecture fails to meet the needs of the software developers are:

- Undefined authority, undefined responsibilities
- Architecture team works in an "ivory tower"
- Not focused on design
- Imbalance in the team composition
- Procrastination

Let us examine some of these traps and how to avoid them.

5.1 Lack of authority

Another story to illustrate the importance of authority.

> A complex telecommunication system started without much of a software architecture. After some time, an architecture team of talented people was created. But the group leaders of the development organisation—the "barons" as the architects called them—took this innovation as a serious challenge to their position and authority. Therefore they ignored whatever was coming out of the software architecture effort, protected what had been their "turf" for a couple of years now and blocked most of the communication between developers. Things did not progress well, the architects became tired and disillusioned and management—unfamiliar with the concept of a software architecture team—did not provide much support. The architects then left the company one after another. The "barons" had won, but the project was now two years late, and still without much of an architecture.

Defining the exact extent of the architects' authority is even more important when a consultant or an external organisation is fulfilling this role.

5.2 Ivory tower

We met the software architecture team of the large multinational company in various public events, and liked their views on software architecture. A few months later, we were called to help one of their divisions and we referred to the company's architecture group. The division management had never heard of the group.

There are other simpler ways of developing the ivory tower syndrome:

- Put the team in another building
- Present software architecture as some kind of sinecure for ageing, or weary designers

The best way to avoid this is to communicate, communicate, communicate. The architecture team, especially when recently created, must make its activity visible by publishing a partial draft of a software architecture document, designing notes, and inviting other people to contribute or review. Every week they should be interacting with the rest of the development organisation.

5.3 Imbalance

An unbalanced software architecture team may have difficulties producing a complete and balanced architecture. A lack of understanding of the domain may lead to an architecture that solves computer science problems only; the same is true if the speciality or main field of interest or experience of the architecture team dominates.

On a project that had no clearly defined software architecture team, we were trying to find out what the architecture of the system was in order to assess it. Interviewing various groups, we got four totally different 'architectures', each group claiming that it was in its charter to define the architecture (cf. Fig. 4).

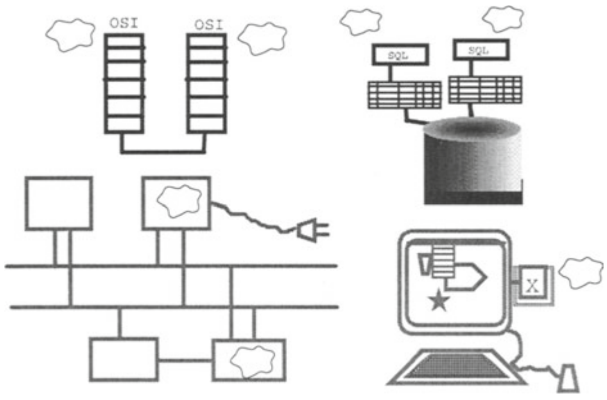


Figure 4. Four specialities, four architectures

1. The telecommunications specialists told us that the main characteristic of the system was its distribution over a vast wide-area network, and that the telecommunication aspect was driving everything. They described OSI protocol stacks, and mentioned some 'code' to be developed as application services elements, hooked at OSI layer six and above.
2. The data engineering leader described the system as one huge database (with impressive E-R diagrams to support this) described the commercial database at the core of the system, and how everything could be managed

by the database, networking, computer-human interface, with only a few algorithms that would have to be programmed in some other language than SQL.

3. The Computer-Human Interface group had done some in-depth study of ergonomic issues, and claimed that the single biggest differentiator between the system under development and its predecessor was in the Graphical User Interface. They spoke about X servers and clients, revolutionary widgets and gadgets, as well as new peripherals.
4. Finally, the systems engineers had a view entirely centred on the computers and the networks. Software was only a strange ingredient you sprinkle on top of the boxes, once you have defined them.

It was hard to believe that they spoke about the same system. Moreover, they were not very concerned about the lack of architecture. Each group had enough to satisfy its concerns in terms of architecture as they defined it.

Morale: find people who have broad experience. Learn about the fields that are under-represented. Eventually bring in specialists to consult with the architecture team.

5.4 Confusing a tool and the architecture

This story also illustrates another danger, which is to acquire a tool or a component that plays a major role in the architecture, and then being led to believe that the architectural design is done—that the tool has defined the architecture.

Vendors of certain major products, especially databases or GUI, would like you to believe that they provide a complete architecture so that once you have bought their product your architectural design is done, that everything will gravitate around their product.

5.5 Procrastination

Attendre d'en savoir assez pour agir en toute lumière, c'est se condamner à l'inaction. *Waiting to know enough to act in full light is to condemn oneself to permanent inaction.* Jean Rostand, French biologist

Procrastination is the worse trap of all; it is insidious, and the best teams can easily fall in it. We have already written that the practice of software architecture is a long and rapid succession of suboptimal tactical decisions, mostly made in partial light. There is a great tendency (especially with people who have scientific training) to want to analyse more, find more options, go further down the paths in order to make the "right" choice, the optimal choice. This slowly kills the project. When decisions are not made, other teams cannot make any progress, or their progress is in jeopardy. They will sit on their hands, waiting for the architects to decide or lose confidence in the architects and make their own decisions.

A project was selecting a tool set to build one of its major components—its graphical user interface (GUI). Several candidate tools were studied: each had its advantages and disadvantages, none was a perfect match. The team delayed the choice, hoping for a better product to appear or for one of the three products to make some significant progress. Meanwhile, the development organisation in charge of the Computer-Human Interface could not make any progress as too much of its work was held back by the lack of tools. Then some hardware choices were not made, because of the lack of understanding of the consequences on the software. It came to a point where the whole project started to suffer significantly because a choice was not being made. The differences between the various GUI products were minimal. Drawing the winner out of a hat would have been better than waiting. But this was not rational. More requests for proposal were issued, more evaluation copies bought, more studies commissioned to establish more criteria for the choice and so on. The sad conclusion of this story is that the final choice was made outside of this team, on a purely political basis, with no consideration whatsoever for any of the technical arguments.

We have found that, in many respects, it is better to make a decision now, in the dark, explaining clearly that it was made with little knowledge of the consequences, rather than suspend a whole project for weeks. Then it is up to the architects watch in the following weeks and months to see if the decision brings more trouble or solutions. If it really becomes a problem, then do not hesitate for one minute: change it. Do not compromise or tergiversate, be very decisive. But do not leave a known evil for another that you do not know, yet. You have put the development back on track. Explore your alternatives, including the cost of changing tracks.

Le courage consiste à savoir choisir le moindre mal, si affreux qu'il soit encore. *Courage is knowing how to choose the lesser evil, as awful as it still is.*
Stendhal

Do not become too focused on the technical optimality of the solution; there is rarely an optimum when taking into account all factors, cost and schedule included. The relative advantages of this or that solution are often minor, and are not enough to justify important delays.

This ability to rapidly make tactical decisions and live with the associated anxiety is one of the elements that distinguishes the software architect from other software developers. It takes a while to get used to it. Some people never do it, and will always hide behind an architect that has more courage than themselves. *The life of a software architect is a long and sometimes painful succession of suboptimal decisions taken partly in the dark.*

Finally, it is very hard to eradicate the idea that software development is not a linear process: it proceeds by trying out ideas to validate them. Therefore, it is OK to start coding things before extensively studying every detail on paper. It is also better to make a choice now (even though it may be the wrong one) and discover early in the project that it is wrong, than to wait forever for the ideal, complete, perfect answer to fall from the sky.

Architects can act as the "conscience" of the products. When management appears to be procrastinating on key issues affecting the product, the architects need to prod them to get the decisions made early. Architects also need to tell management the bad news, including negative results and failed prototypes, early enough so that adjustments can be made.

6. THE PERSONALITY OF THE ARCHITECT

Is there a certain psychological profile that suits the role of the software architect? Maybe. Based on original ideas of the Swiss psychologist Carl Jung, American psychologists Isabel Myers and David Kersey developed, in the 50's and the 60's, a classification of personalities which has had some success in corporations throughout North America, most notably under the label "Myers-Briggs Type Indicator." They developed psychological tests to classify individuals according to four major traits:

1. Extroversion (E) versus Introversion (I)
2. Sensation (S) versus Intuition (N)
3. Thinking (T) versus Feeling (F)
4. Perceiving (P) versus Judging (J).

Although these four characteristics are not binary, but rather a scale—one can be extroverted to some degree—one may classify individuals in 16 "bins" of irregular size, labelling each bin with the letters indicated above: ESTP, ISFJ, etc. The type ESTP would therefore describe a person whose personality leans towards Extroversion, preferring Sensation over Intuition, relying on Thinking more than on Feeling and using Perceiving rather than Judging. Over three decades psychologists studied common characteristics of each of the 16 groups of individuals, notably how they fit in their working environment. They refined the model to introduce "mixed types," taking into account the traits where the test does not clearly lean towards a letter or another, marking this with the letter X, such as EXTP for someone who would be in between ESTP and ENTP.

All this preamble is to tell you that it seems that good software architects are found among the INTJ or INTP types. Not much surprise about the 'I':

most people in scientific or technological fields are introverted. The NT part is the Promethean temperament: the 12% of the population who loves intelligence, power over nature, competence, skills, and their work. NT types like to be liked for their ideas. When in a leadership position, they are visionary leaders. They do not like routine. They are the vectors of change.

David Keirsey nicknames the INTP the "Architect", the "Abstractionist". Abstract design is their forte and coherence is the primary issue. They are curious, rational, and theoretical. The world exists to be understood. They are the logicians, the philosophers of systems. They exhibit a great precision in thought and language. They easily detect contradictions and flaws. But they can also become obsessed with analysis or the gathering of more data.

Keirsey nicknames the INTJ the "Scientist." INTJs are the most self-confident of all 16 types, with a great awareness of their own power. Authority or slogans have little impact on them, unless it makes sense. They can easily make decisions, bringing issues to closure. Unlike the INTPs, they need only to have a vague, intuitive impression of the unexpressed logic of a system to continue surely on their way. They rapidly discard theories that cannot be made to work. They are better at generalising, classifying and demonstrating than INTPs. They are less likely to procrastinate.

Although we do have some very limited evidence that successful architecture teams are primarily composed of INTJ and INTP, other types are useful to achieve a good balance as the team grows: for example an ISTJ—the highly dependable "trustee"—would keep track of things in a large project. While some extroverted types could improve communication. Thus, we would satisfy one of Katzenbach and Smith's axioms for a team: a blend of technical and functional skills, problem-solving and decision making skills, and interpersonal skills. The bad news is that INTJs and INTPs represent only 2% of the general population. After selecting people based on their expertise, you may not have much latitude left, unless you are in a big company with deep pockets.

7. SUMMARY

- Designate a software architect, or assemble a small team of software architects who share a common goal or vision of the product.
- The software architecture team must be experienced in both the problem domain and software development.
- Software architects should be fully dedicated to their task; in particular, the role of software architect is usually not compatible with that of project manager.
- The architect(s) and the project manager are joined at the hip.

- Establish a charter of the software architecture team which clearly defines its role and responsibilities, and establishes its authority.
- Do not isolate the software architecture team—it is a software development group among other software development groups.
- Common pitfalls for a software architecture team include: lack of experience, undefined authority or isolation, an unbalanced mix of technical skills, lack of focus on the design, and procrastination.

REFERENCES AND FURTHER READING

Grady Booch, *Object Solutions*, Addison-Wesley, Menlo Park, CA, 1996.

Frederick P. Brooks, Jr., *The Mythical Man-Month—Essays on Software Engineering*, 2nd edition, Addison-Wesley, Reading MA, 1995.

Carl Jung, *Psychological Types*, Harcourt Brace, New York, 1923.

Jon R. Katzenbach and Douglas K. Smith, *The Wisdom of Teams*, Harper Business, New York NY, 1993. They give good examples from business cases of good, great, and not-so-good teams. Then they extract from their examples the underlying principles of what makes teams tick and become "high performance organisations".

David Keirsey and Marilyn Bates, *Please Understand Me—Character and Temperament Types*, Prometheus Nemesis Book Co., Del Mar, CA, 1984. A very practical and easy-to-read explanation of the Myers-Briggs classification, and its use in everyday life, at work and elsewhere.

John A. Mills, "A Pragmatic View of the System Architect," *Comm. ACM*, 28 (7), July 1985, pp. 708-717. In this very lively paper, Mills describes the roles of the system architect: "A whole-system designer, fire-fighter, mediator, and jack-of-all-trades, the system architect brings unity and continuity to a development project—offsetting the inevitable compartmentalisation of modern modular designs." He depicts three slightly different organisations each with a system architect or a team of system architects. He also justifies the necessity of a central architectural function when a project reaches a certain critical mass, so that the number of one-to-one communication links between the various development groups is reduced.

Isabel Myers, *Manual: The Myers-Briggs Type Indicator*, Consulting Psychologists Press, Palo Alto, CA, 1962. She describes the 16 types and the associated assessment procedure.

Eberhardt Rechtin, *Systems Architecting: Creating and Building Complex Systems*, Prentice-Hall, Englewood Cliffs NJ, 1991. Chapter 14 (p.289-293) describes the profile of a system architect, as does the following paper.

Eberhardt Rechtin, "The Systems Architect: Specialty, Role and Responsibility," *Proceedings of NCOSE*, 1994.

Mary Shaw & David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996. Chapter 9 speaks about the Education of Software Architects and describes the course being taught at CMU.

Aspect Design with the Building Block Method

Jürgen K. Müller
Philips Research Laboratories, Prof. Holstlaan 4 (WLp), NL - 5656 AA Eindhoven
mueller@natlab.research.com

Key words: Software architecture, software component, object-oriented modelling, function-oriented modelling, design method

Abstract: Aspects are a way to supplement object-oriented modelling with function-oriented modelling. The Building Block Method (BBM) identifies 3 dimensions of independent design. Besides an object dimension and a process dimension also an aspect dimension is present. Object design, process design and function design are done independently. Building Blocks (BB), which are software components, cluster functionality according to criteria such as configurability in a product family and incremental system integration. The BBM is used to design families of telecommunication infrastructure, digital broadcasting and medical imaging systems. The paper introduces the concept of aspects and shows how they are identified and used in the BBM.

1. INTRODUCTION

Today's electronic systems implement more and more of their functionality in software. The flexibility of software and the price erosion of standard computing hardware further this trend. Despite all kinds of modularity in hardware and software, the integrating system characteristics of the larger systems are always implemented in software. These architectural software structures are much harder to change than more local hardware and software parts. The continuous evolutionary development requires that software and hardware needs to be changed and extended in a piecemeal way. To enable this the basic architectural structures must be designed to do this with moderate effort.

The use of object-oriented modelling has advanced the conceptual level of implementations as consisting of a network of objects. For small and medium size systems this may be enough. Large systems, however, easily become monolithic, i.e., an unmanageable web of objects. They therefore need more modularity and more locality of changes in an evolutionary development situation. Object-oriented concepts should be complemented by other concepts from other modelling techniques, for example functional modelling. The goal is to create systems that are modelled naturally for their application domain [10]. BB aspect design is a means to cope with this problem.

The next section gives an introduction to the BBM. Section 3 defines software aspects and shows by means of examples how software aspects are derived from a system level perspective. Software aspects are an important means to relate software functionality to the overall system functionality. Section 4 and 5 look at the consequences for components and component-based development. Section 6 compares the aspects in the BBM with other approaches.

2. OVERVIEW OF THE BBM

The BBM is a component based [18] design method for the development of the software for central controllers of embedded systems. These central controllers integrate, control, and manage the overall system. They are points of great complexity.

Many of the concepts that we will present can (or better, should) be used for all of the software of an embedded system; in fact they are useful in the structuring of *any* large software system. However, we focus on the specific problem of central controllers to be able to reason very specifically, which would be more difficult if we dealt with software systems in general [9]. Furthermore, the basic ingredients of the method presented have been tried out successfully in the design of these central controllers.

Experiences with the method stem from the development of systems in the telecommunications, video broadcasting and medical imaging domain. The most complete overview of the BBM can be found in [12]. Some quantitative figures are given in [16].

The method presented is designed to support the creation of product family architectures [4]. Composing a product from pre-manufactured components is how software reuse in a product family architecture is achieved [13]. The product management requirements of short lead time, low development effort, and high quality products have been translated into the architectural requirements of conceptual integrity, managed complexity, delta development, extensibility, reusability, configurability, and testability [12]. These requirements have been taken as the main ones to be achieved in a system architecture constructed via our method.

However, it is clear that architecture can only lay a good foundation on which the system is built. A good product needs more than just a good architecture. Errors can be made at all levels of system development: requirements, architecture, detailed design, implementation, deployment, documentation, to name the most important ones. In the next section we take a look at the architectural meta-model (AM-M) of the BBM.

2.1 The AM-M of the building block method

Traditionally the description of an architecture and its models have been very much dependent on the persons presenting them. Depending on the major problems to be solved by the architecture, or the personal preferences of the persons involved, structures like hardware boundaries, modules, processes or state machines have been presented as *the* architecture.

However, experiences from different projects show that besides the specific nature of a product the architecture had to address a number of common problems. These problems are, for example, the units in which a software system is broken down, composed and evolved; or the way in which processing resources are to be used by the application. The solutions to those problems are part of an architecture.

Different methods define different things to be part of an architecture. We call those parts of an architecture that are required by a method the architectural meta-model of a method.

The architectural meta-model of the BBM consists of

- a logical model,
- the building block design dimensions (BBDD), and
- components, the so-called building blocks (BB).

We explain the AM-M of the BBM in the rest of this section.

2.1.1 Logical model

The logical model is the model of the functionality of the systems to be built. It describes the externally observable (and the made-known internal) functionality. The language used is the one of the customer (or user) and of the product managers. The intention of the model is to precisely describe

- the "what" of the system.
- the environment of the system, e.g., to which interfaces it has to comply.
- other conditions for the system.

The "how" of the implementation is left to design.

2.1.1.1 Modelling language and domain modelling

A lot of different modelling languages have been proposed. Examples include data flow diagrams, entity-relationship diagrams, state machines, object-oriented modelling. However, no standard has yet evolved, and it is questionable if that will ever happen.

If you look at a very mature domain, say cars, a lot of domain-specific objects are used to describe and compare cars. Also new cars are described in terms of the attributes of domain objects, e.g., the number of cylinders of the motor and its horse power, the type of gearbox, the interior design, its maximum speed, etc. In a new domain a description of a product uses many more functions of that product than attributes of domain objects. The objects are still a matter of design.

However, in many domains object-oriented modelling will be the modelling language of choice. The most prominent one is the Unified Modelling Language UML [6]. There are many methods to develop such an OO model.

As a domain matures and companies want to cover a complete application domain with their products the focus of the logical model changes. Being initially the model of the functionality of a product, the logical model is used to describe a complete application domain (together with a selection list for each specific product).

The BBM does not use a specific modelling language nor a specific method to develop such a logical model. It assumes that one exists and takes it as basis for the further architectural design.

2.1.2 Design dimensions

The idea of dimensional structuring is introduced to support intellectual control of system functionality. If common concepts or structures that apply to the same item can be separated such that there are no mutual restrictions, the concepts or structures are orthogonal; we can talk about design dimensions. Thus, design can be carried out independently for every dimension. Each item in the design space can be reduced (projected) to one specific dimension. The BBM identifies 3 design dimensions that will be motivated below. Of course functionality of the 3 dimensions has to be related. Design guidelines for each dimension and for relating the 3 are given. However, the BBM views every such relation as a design decision.

The first point is to separate object structuring from the use of execution units. Objects stem from the modelling of domain functionality. Execution units determine the use of processing resources for independent, co-operating and/or sequential actions. The designer should be free to do the process design without consequences for the object design. They build two orthogonal dimensions, i.e., a method of an object may be driven by one or more processes and a process may drive methods from different objects.

The second point is to introduce functional structuring in addition to object-based structuring. Those global functions that deal with potentially all objects (i.e., they cross-cut objects) are handled as *aspects*. Thus each object method in the system is part of an object and part of an aspect.

The BBM combines these two points, which leads to three design dimensions: the object dimension, the aspect dimension, and the process dimension (figure 1).

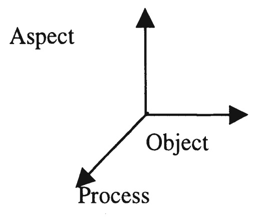


Figure 1. Design dimensions

In the BBM, an object method is defined to be the minimal block of functionality. This corresponds to a point in the design space. The first dimension is the object dimension, which covers the decomposition of the system into data units with their access procedures. The second dimension is according to aspects. Each aspect deals with a specific view of the functionality of the systems, such as recovery, configuration management, fault handling, etc. The whole system functionality is partitioned by the set of aspects. The third dimension is the process dimension, viz. the whole system is driven by a set of processes. The guidelines to identify and define objects and processes are not the subject of this paper.

The consequence is that design for each dimension can be made independent of the other two, which gives the freedom to make a design that is best suited for the application. The structuring in the design dimensions is a structuring at the meta-level of the system.

2.1.2.1 System evolution and design dimensions

With respect to system evolution, the aspect dimension is not handled in the same way as the object dimension. We assume that a system is primarily modelled using objects. For most application areas this gives the most stable modelling. The most common extensions of the system are via extended objects and new objects (i.e., there exists a locality of change).

The two design dimensions, object and aspect, are a specific union of object-oriented and functional modelling. Aspects are seen as a secondary form of modelling. They are functions that cross-cut all or most objects. To achieve stable software structures aspects should be standardised for a complete product family. Adding a new aspect affects all related objects.

Process design starts by looking for independence of objects. But for most systems we assume that a situation is aimed for where the structure of independent execution units is standardised, either in specific classes or in some rules that guide the creation of classes and instances (see Gomaa [7]). Without such rules the understanding of a large evolving system is very difficult.

Because of the above-described situation, that aspects and processes should be stable, we say that the evolution of the system is mainly in the object dimension.

2.1.3 Components

Components are deployment units that are identified in the architectural phase [18]. This means that they are present during all of the development phases. They are called Building Blocks (BB). They are the main focus of the BBM. An architecture identifies the BBs. A specific product is built out of BBs. This means that a BB has a specific representation in all the phases.

There are two important points for BBs: what is the content of a BB and what are the relations between BBs? Through the design dimensions the functionality of a complete system is designed. A BB usually consists a collection of objects. It does usually not contain complete processes or aspects. The set of BBs covers the complete functionality in a non-overlapping way. There are certain criteria for identifying BBs; the main one is configurability. These criteria may also influence the design along the 3 design dimensions. A design process evolves typically in several steps until stability is reached.

A system architecture has to define inter- and intra-BB structures. Aspects are used for intra-BB structuring (see below). Inter-BB structures are defined via the concepts of layers, subsystems, classification of BBs, and a skeleton.

The include relations of all BBs form a partial order. Each BB resides in a layer [3] and can only use BBs in strictly lower layers. During the initialisation phase a BB in a higher layer binds itself to a BB in a lower layer (post-load linking). Through this binding BBs in a higher layer establish call-back procedures at lower-layer BBs. This guarantees that on the syntactic level no mutual relation exists. Design rules exist that avoid that on the semantic level.

BBs are classified into generic and specific BBs. Generic BBs implement the generic part of some functionality. Specific BBs implement the delta part of that functionality. The classification in generic and specific functionality is a relative one. A BB is generic or specific with respect to a specific functionality. A BB can be generic and specific with respect to several functionalities. Examples of infrastructure functionality are a device driver abstraction vs. specific device drivers, a BB implementing a fault handling concept vs. those implementing specific faults, etc. One generic BB has usually several specific BBs. A generic BB is always located in a lower layer than a specific BB, i.e., a specific BBs uses functionality of the generic BB.

Several layers of BBs can be grouped into subsystems [5]. Subsystems follow the rules for layers: subsystems are stacked above each other. To allow configurability even in lower subsystems some of the BBs have only "requires" interfaces, i.e., their functionality is only accessed via call-back interfaces. They can be added and removed without syntactical consequences for BBs in higher subsystems. Subsystem access generics connect functionality from higher subsystems with those configurable BBs.

3. SOFTWARE ASPECTS

Software aspects are global functions that cross-cut domain objects. To identify aspects we have to look at a functional structuring of the application domain. Relating these functional structures to the identified objects can lead to 3 cases:

1. a function has relations to, and/or defines functionality of, a few objects only,
2. a function is to be used by almost all objects,
3. a function defines the functionality of almost all objects.

In the first case the function will be handled as (part of) some functional object of the object dimension. In the second case the function will also be handled as a functional object of the object dimension but will be part of the system infrastructure. In the third case the function will be an aspect. Such a function cross-cuts objects. Aspects are a non-hierarchical, potentially complete, functional decomposition of software functionality. The list of aspects should be anchored in the application domain and be defined for a complete product family.

The decision to model such a function as aspect or not depends on the required functionality. Let us take the function of access control as an example. If access control is to be done whenever a user wants to enter a system and, if access is granted, the user is free to use all functionality, access control can be localised as an "access control object" that implements all required functionality. On the other hand, if access control should be more sophisticated depending on users and user groups that have certain rights at certain times, the functionality may logically belong to the application objects. A design may use access control lists and a state model for each object to decide if access is granted. Access control could be defined as an aspect of all objects. An implementation could be split into a generic access control object that implements all common functionality, while any other object has to implement its specific access control functionality. The generic component would be part of the system infrastructure. This example shows how the second and third case of function object relations can sometimes be related. A file system is a simple case of such a system where access rights are located with the files while the processing of the access rights is handled in the "file handling object."

To identify aspects we have to look at the functionality from a system perspective. We give several examples of ways to look at system functionality. Some are general, while others stem from specific domains. From these views we derive system aspects. In a further step, ways to derive software aspects from system aspects are described. The identification of aspects has to be done for a specific domain, as it is with objects.

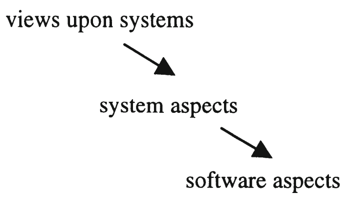


Figure 2. Deriving software aspects

3.1 Aspects of system design

System design is a multi-stakeholder and multi-disciplinary task. Besides the functional requirements of the system, non-functional requirements of the customer and the developing organisation have to be met. These different views of a system's functionality constitute the concerns of system design. An architect has to take all these concerns into consideration.

In the following, four different sets of views of a system are presented. They are from different contexts and we do not discuss their pros and cons; they are used as a starting point for the definition of software aspects. In a first step, however, we classify these views according to whether they are directly relevant for system implementation or whether they only shape the context of system implementation. We call views that directly influence system implementation system aspects (figure 2).

3.1.1 Functional and non-functional requirements

Requirements are often classified into functional and non-functional requirements. The intention of this classification is to emphasise the fact that besides functional characteristics many more qualities are expected from a well-designed system. Depending on the system and also the customers, system attributes such as performance, safety, technology choices, testability, reuse, portability, use of standards, etc., are part of the customer requirements or not. A customer can specify these requirements either not at all, partially, or fully. Implicit system attributes that are expected to be present in all systems of a certain class in a specific market segment have to be added. A development organisation will add requirements because of internal benefits or consistency. Therefore the classification into functional and non-functional requirements is, in general, somewhat vague. It cannot directly be used to guide a design. System aspects cover functional and non-functional behaviour.

3.1.2 System quality attributes

Quality attributes are another view upon the system. Bass et al. [1] classify system qualities in 4 classes:

1. business qualities, such as time to market, cost, projected lifetime of the system, targeted market, roll-out schedule, extensive use of legacy systems
2. quality attributes discernible at run time, such as performance, security, availability, functionality, usability
3. quality attributes not discernible at runtime, such as modifiability, portability, reusability, integrability, testability
4. intrinsic architecture qualities, such as conceptual integrity, correctness and completeness, buildability

These qualities are intended to guide the process of architecting a system. Business qualities determine the context of system implementation. Quality attributes discernible at runtime and those not discernible at runtime are system aspects. Intrinsic architecture qualities guide the process of making an architecture but do not directly influence system implementation.

3.1.3 Architectural concerns

G. Muller made a list of architectural concerns [14] for the design of medical imaging systems. He made the point that the system architect has to take all these concerns into consideration (i.e., know the specific requirements, communicate with the respective stakeholders, judge on the relative importance, etc.). The architectural concerns are:

- application requirements *,
- functional behaviour *,
- typical load *,
- resource usage (CPU, memory, disk, network, etc.)*,
- installation, configuration, customisation, etc. *,
- factory and field testability *,
- configuration management (technical and commercial) *,
- safety, hazard analysis *,
- security *,
- image quality *,
- functional chain specifications (print, store, etc.)*,
- interoperability, other systems, selected partners, other vendors *,
- interfacing to other applications *,
- technology choices (software, hardware, computer, dedicated digital, make/buy),
- selection and use of mechanisms,
- module design, process design, function allocation (method, file, component, package)*,
- information model: world standardisation, PMS standardisation, PMG standardisation, application specific *,
- test strategy, harnesses, suites, regression,
- verification *,
- performance, throughput, response *,
- re-use consequences, provisions; development process impact; organisational impact; business impact,
- assessment of strong and weak aspects, road map for all views,
- system engineering (cables, cabinets, environment, etc.),
- cost structure (material, production, initial, maintenance, installation),
- logistics, purchasing (long lead items, vulnerability, second sourcing).

These architectural concerns are broad. They look at the system (to be built), its development and use environments. A more restricted view is to look only at system aspects. Architectural concerns that are system aspects are denoted with an asterisk.

3.1.4 Operator oriented system aspects

In the area of telecommunication infrastructure, systems tasks and procedures of operators have been classified. A system has to provide interfaces and functionality to support an operator in his or her tasks. A traditional classification distinguishes operation, maintenance, and administration tasks; it is often abbreviated OMA. FCAPS is the classification of the OSI system management functional areas (SMFAs) [8]. The functions are divided into fault management, configuration management, accounting management, performance management, and security management. These operator oriented function classifications are system aspects.

3.2 Mapping of system aspects to software

Software aspects are derived from system aspects. To do this we first classify system aspects in those that directly specify functionality and those that put constraints on how functionality is implemented. Operator oriented system aspects specify functionality, while system aspects from the system qualities constrain the implementation of functionality. System aspects from the architectural concerns list fall in both classes. Some of them specify functionality that is realised in software. The relation between system aspects and software aspects can be described by the following mappings:

- not relevant for software (e.g., handled in hardware)
- mapped to functional unit (e.g., domain object, BB, subsystem)
- mapped to own software aspect
- mapped to shared software aspect
- distributed over several other software aspects and/or functional blocks

In the following, three examples are given to describe this process. As a first example the aspect list of the tss system where the BBM has been applied first is given. The second example describes the rationale of mapping a system aspect. The third example gives a list of software aspects derived from the architectural concerns list.

3.2.1 Example: tss software aspects

As an example we give the list of software aspects in the tss system [2].

- system management
  The aspect system management deals with the external control of the system. This may be a man-machine interface, including formatted input and output, or a message-based coded interface.
- recovery
  The aspect recovery deals with the proper initialisation of the system during recovery time.
- configuration control
  The aspect configuration control deals with the impact of changes in the physical (hardware) and/or logical configuration; changes may have been induced by failures or reconfigurations via system management.

- data replication
  The aspect data replication deals with the replication of data across processor boundaries. Configuration data of the controller are replicated whenever a peripheral device requires a local copy of part of the configuration data.
- test handling
  The aspect test handling comprises built-in functions running periodically, or being invoked on specific events, in order to detect and identify internal or external hardware faults or corrupted data. Test functions have no resulting event except to indicate a failure.
- error handling
  The aspect error handling is entered when a failure occurs. The related functions take the appropriate actions on a failure. This especially includes damage confinement and fault localisation.
- diagnostics
  Functions of the aspect diagnostics are invoked by
  - test handling in order to detect faults in the sense of preventive maintenance,
  - error handling in order to localise hardware faults,
  - configuration control in order to verify the repair or re-configuration of physical or logical objects.
- performance observation
  The aspect performance observation deals with the collection and processing of data for statistical and quality measurement purposes.
- debugging
  The aspect debugging covers the functions required to debug the on-line software in test-floor operation as well as filed operation.
- overload control
  The aspect overload control implements the functionality to prevent the system from being overloaded. During the overload situation the system is still within the margins of the specified quality of service.
- operational
  The aspect operational has a specific character. It represents the core functional behaviour of the system, i.e., handling of calls.

3.2.2 Example mapping of system aspect reliability in tss

To illustrate a mapping of a more complex system aspect the implementation of the system aspect reliability in the tss switching system is given.

Reliability is realised in the following ways:

- not in software:

  - the central processor is in a 1:1 redundant configuration that operates in cold stand-by mode
  - for the 3 classes of peripheral cards the following holds:
    - specific service cards are configured in load sharing or hot stand-by for dynamically allocated resources
    - the network of switching systems implements hot stand by of trunk cards
    - subscriber cards are not redundant
- software aspect man-machine interface: changes to the card configuration, states of card and logical objects and parameters thereof made by the operator have transaction semantics
- software aspect configuration: persistence of the configuration is realised in a database
- software aspect error handling: fault management concepts are implemented for card faults to hold the system in a consistent state

3.2.3 Example of a hypothetical system

As another example we present a possible definition of software aspects for a hypothetical medical imaging system. The definition is based on the list of system aspects given by G. Muller (see above). A system could have the following list of aspects:

- operational (functionality to process medical images)
- initialisation and recovery
- pixel data communication
- control communication
- imaging information model
- configuration management
- safety and consistency checking
- user interfacing
- testing

These aspects represent functionality that is orthogonal to object modelling. Another concern such as logistics may lead to the use of an identification number for BBs. Comparable cases can be derived from other concerns as well.

4. ASPECTS AND BUILDING BLOCKS

Most of the systems designed with the BBM have their features in the object dimension. That means that most of the BBs contain one or more objects completely. Therefore for each BB a standard substructuring has been defined: all aspects are present in each BB, even if some of the aspects are empty for a BB (figure 3). Note that some of the aspects such as debugging may require functionality to be present everywhere. This could mean that a BB automatically has to implement that functionality. Functionality for another aspect such as error handling is only present in parts of the system where errors can occur (virtual fault absence on higher layers).

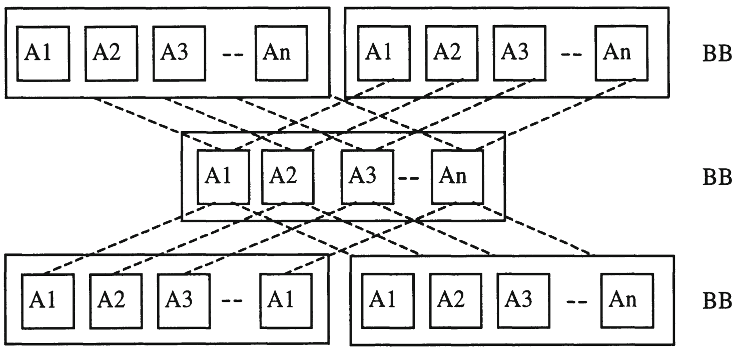


Figure 3. Aspect structuring of building blocks

That means that systems designed with the BBM have BBs as their primary decomposition, with secondary structuring provided by the aspects within the BBs. Introducing new aspects for a system is not forbidden, but it is a worst-case situation where the change effort is almost maximal.

5. ARCHITECTURAL DESIGN WITH ASPECTS

The list of aspects is a tool for the architects to check the functional completeness of their identified components. Questions such as: which initialisation actions are required by a component, which faults can a component have, how can it be influenced by other faults in the system, how can the component be configured, what is the required reliability, which resources may it use, etc., help to specify its required functionality.

5.1 Aspect-completeness of configurable components

Application features are modelled ideally if they can be added to an installed system. Even more advanced is if the complete system consists of pluggable components. To be able to implement systems from components only, these components have to be functionally complete. Functional completeness is relative to a given functional infrastructure. Aspects together with a well-designed infrastructure are a means to achieve this functional completeness. A (set of) BB(s) is aspect-complete if it allocates all its required resources itself and implements all aspect functionality [15].

5.2 Aspects and documentation

In the BB method the notion of a BB is pervasive from architectural design to implementation. In the architectural design BBs are identified, the specification phase completes the specification of a BB's functionality, in the design and implementation phases it is designed and implemented, respectively. The list of aspects is used for completeness checking in review sessions. Each BB has its own documents: here, again, aspects are used as the main chapters of the document.

5.3 Aspects and implementation

In the implementation each function is a triple <object, process, aspect> in the design space, i.e., each function is part of an object, is driven by a process, and is part of an aspect. Making aspects a standard substructuring of a BB, provides a secondary modularity. Naming conventions, files, or programming language modules are possible ways of implementing this modularity. Some of the aspects of a BB may be empty.

6. COMPARABLE APPROACHES

The architectural models of Kruchten [11] and Soni et al. [17] also make the distinction between object and process dimensions for their implementation structuring. Kruchten uses the terms development view and process view, while Soni et al. use module interconnection architecture and execution architecture. The examples given by Soni et al. indicate that the conceptual architecture provides a functional decomposition that is hierarchical to the development and the execution architecture. Kruchten's logical view provides no constraints for further structuring in the development and process views.

Kruchten's model is object-oriented and recognises the independence of the modelling of processing resources. It leaves out the aspect dimension, i.e., functions are subordinate to objects. Soni et al. work with a functional structuring and on the next level distinguish between development units and processes. The functional structuring is dominant; object-oriented structuring may be used on a micro-level. Perhaps this is the case because their model is more a reverse-architecting model than a forward-architecting model.

The logical model of the BBM is close to the one developed by Kruchten [11], however we can also imagine more function-oriented logical models. The original project where the tss system was developed and from which the BBM originates took only a list of features as the logical model. This may be too limited for domains where most of the domain knowledge is not implicitly present. On the other hand the logical model of Kruchten does not support nor hinder feature-list-like descriptions.

Independent of architectural discussions, limitations of the object-oriented design have been recognised. Kiczales et al. [10] describe examples where object-oriented modelling is too limited and leads to very complex code. He is looking for an alternative structuring that leads to a natural design structure also for more complex examples. Kiczales calls his approach aspect-oriented programming (AOP). He defines an aspect to be functionality that cross-cuts objects. Since his concern is programming and development of next generation programming languages, it could be said that he does bottom-up what the BBM method does top-down, from the system point of view. Furthermore, the need to define (sub-)languages in AOP for each kind of problem creates very specific solutions only.

ACKNOWLEDGEMENTS

Thanks to Lothar Baumbauer, who, as chief architect, promoted the consequent use of aspects in the tss development. Thanks also to my colleagues Frank van der Linden, Hans Jonkers, Henk Obbink, René Krikhaar, Angelo Hulshout, Jan Gerben Wijnstra and William van der Sterren for valuable discussions and comments to earlier versions of the paper.

REFERENCES

[1] Len Bass, Paul Clements, Rick Kazman: *Software Architecture in Practise,* Addison-Wesley, 1998

[2] Lothar Baumbauer: *System Level Documentation,* Volume 6014 (internal documentation) Philips Kommunikations Industrie AG, 1995

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: *Pattern-oriented Software Architecture - A System of Patterns*, Wiley and Sons Ltd., 1996

[4] Paul C. Clements: *From Domain Models to Architectures,* Workshop on Software Architecture, USC Center for Software Engineering, Los Angeles, 1994

[5] Paul C. Clements: *From Subroutines to Subsystems: Component-Based Software Development*, The American Programmer, vol. 8, no. 11, November 1995

[6] Martin Fowler: *UML Distilled, Applying the Standard Object Modelling Language,* Addison-Wesley, 1997

[7] Hassan Gomaa: *Software Design Methods for Concurrent and Real-Time Systems,* Addison-Wesley, 1993

[8] ITU: *Management Framework for Open Systems Interconnection (OSI) for CCITT Applications,* Recommendation X.700, September 1992

[9] Michael Jackson: *Formal Methods and Traditional Engineering*, Journal on Systems and Software, vol. 40, pp. 191-194, 1998

[10] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, John Irwing: *Aspect-Oriented Programming,* Xerox Corporation, 1997

[11] Philipe Kruchten: *The 4+1 View Model of Architecture,* IEEE Software, Nov.1995

[12] Frank van der Linden and Jürgen K. Müller: *Creating Architectures with Building Blocks*, IEEE Software, Nov. 1995

[13] Frank van der Linden, Jürgen K. Müller: *Composing Product Families from Reusable Components,* Bonnie Melhart, Jerzy Rozenblit (eds.) Proceedings 1995 International Symposium and Workshop on Systems Engineering of Computer Based Systems, IEEE, pp. 35 Ð 40 (1995)

[14] Gerrit Muller: *Systeem ontwerper een twintig koppig monster?*, personal communication

[15] Jürgen K. Müller: *Feature-Oriented Software Structuring*, CompSAC'97, pp. 552-555, August 1997

[16] Muthu Ramachandran, Wolfgang Fleischer: *Design for Large Scale Software Reuse: An Industrial Case Study*, 4th Inter'l Conf. on Software Reuse, Orlando, Florida, April 1996

[17] Dilip Soni, Robert L. Nord, and Christine Hofmeister: *Software Architecture in Industrial Applications*, ICSE'95, Seattle 1995

[18] Clemens Szyperski: Component Software - Beyond Object-Oriented Programming, Essex 1998

Erratum to: Software Architecture

Patrick Donohoe

Carnegie Mellon University, USA

Erratum to:
P. Donohoe (Ed.)
Software Architecture
DOI: 10.1007/978-0-387-35563-4

The book was inadvertently published with an incorrect name of the copyright holder. The name of the copyright holder for this book is: © IFIP International Federation for Information Processing. The book has been updated with the changes.

The updated original online version for this book can be found at
DOI: 10.1007/978-0-387-35563-4