COMPUTER AIDED SOFTWARE ENGINEERING

HAUSI A. MULLER
RONALD J. NORMAN
JACOB SLONIM

KLUWER ACADEMIC PUBLISHERS

COMPUTER AIDED SOFTWARE ENGINEERING

edited by

Hausi Muller
University of Victoria

Ronald J. Norman
San Diego State University

Jacob Slonim
IBM Canada, Ltd.

A Special Issue of
AUTOMATED SOFTWARE ENGINEERING
An International Journal
Volume 3, Nos. 3/4 (1996)

KLUWER ACADEMIC PUBLISHERS
Boston / Dordrecht / London

AUTOMATED SOFTWARE ENGINEERING

An International Journal

Volume 3, Nos. 3/4, August 1996

Special Issue: Computer Aided Software Engineering
Guest Editors: Hausi A. Muller, Ronald J. Norman and Jacob Slonim

Distributors for North America:
Kluwer Academic Publishers
101 Philip Drive
Assinippi Park
Norwell, Massachusetts 02061 USA

Distributors for all other countries:
Kluwer Academic Publishers Group
Distribution Centre
Post Office Box 322
3300 AH Dordrecht, THE NETHERLANDS

Library of Congress Cataloging-in-Publication Data

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

ISBN-13: 978-1-4612-8626-4 e-ISBN-13: 978-1-4613-1439-4
DOI: 10.1007/978-1-4613-1439-4

Printed on acid-free paper.

Automated Software Engineering 3, 189–191 (1996)
© 1996 Kluwer Academic Publishers. Manufactured in The Netherlands.

CASE: The Coming of Age for Automated Software Development

HAUSI A. MÜLLER
University of Victoria

RONALD J. NORMAN
San Diego State University

JACOB SLONIM
IBM Canada Ltd.

Over the last dozen years, computer-aided software engineering (CASE) has continued to evolve and improve, but so has the state of software development research and its application in the commercial marketplace. What James Martin, in 1989, called "industrial-strength" CASE, was almost entirely PC-based. It was not even close to the two- and three-tier CASE architectures of 1996.

As software development moved into the 1990s, collaborative CASE and integrated CASE environments were beginning to appear in software development organizations. However, in the last five years, the quick and steep rise in demand for GUI-based software has provided numerous challenges and opportunities to CASE researchers, vendors, consultants, and practitioners.

The target that CASE is attempting to hit—large-scale software development—continues to move as industry adopts advances in technology. That creates a constant challenge to CASE acceptance in the marketplace. To face that challenge, CASE '95, the Seventh International Workshop on CASE, convened in Toronto, Canada in July 1995. Over 200 researchers, vendors, and practitioners met to assess the current state of CASE; review and discuss current CASE-related research; preview, sell, and research CASE tools; and discuss the direction CASE should take for the next few years. Much work was accomplished under the direction of Jacob Slonim, Head of Research, IBM Canada Ltd., who served as General Chair for CASE '95.

The CASE '95 International Program Committee, under the direction of the Program Co-Chairs Hausi Müller and Ronald Norman, worked diligently to assemble forty high-quality research papers, interesting tutorials presented by leaders in their fields, and workshops on important and timely topics. There were over 120 research papers submitted from around the world. The first Stevens Lecture on software development methods in honor of the late Wayne P. Stevens was given by Tony Wasserman, founder and chairman of Integrated Development Environments, Inc. (IDE), USA.

The eight research papers selected for this special issue of JASE were rated highly by the CASE '95 International Program Committee, subjected to JASE's rigorous review standards, and substantially revised since their inclusion in CASE '95's Proceedings. Each paper is briefly introduced here.

As its title suggests, "Automating the Software Inspection Process" by MacDonald, *et al.* reviews four main areas of software inspection automation being utilized today-document handling, individual preparation, meeting support, and metrics collection. That overview precedes a description and comparison of five tools that have been developed to support the inspection process. The authors summarize by discussing the additional features and associated benefits that could be provided by automated support for inspection.

Gangopadhyay and Mitra's article, "Design by Framework Completion," explores the notion of exemplar, which they define as an executable visual model for a minimal instantiation of the architecture. An exemplar documents frameworks that define an architecture for a family of domain-specific applications or subsystems. This article proposes a paradigm shift when designing in the presence of reusable components. The authors advocate a top-down approach for creating applications in which all components obey the same architectural rules that are governed by the framework.

The third article, "Building an Organization-Specific Infrastructure to Support CASE Tools," by Henninger advocates an organization-wide development infrastructure based on accumulated experiences within application and technical domains. The domain life cycle formalizes a process for accumulating project experiences and domain knowledge, thus freeing the developers to concentrate on less well-known elements of an application.

Ng, Kramer, and Magee's article, "Automated Support for the Design of Distributed Software Architectures," describes a "software architect assistant," which is a visual tool for the design and construction of distributed systems. Their tool supports a compositional approach to software development. Their objectives for the tool are to automate mundane clerical tasks, enforce program correctness and consistency, and accommodate the individual working styles of developers.

"Domain Modeling for Software Reuse and Evolution" by Gomaa, *et al.* describes a prototype domain-modeling environment used to demonstrate the concepts of reuse of both software requirements and software architectures. Their environment, which is independent of the application domain, is used to support the development of domain models and to generate specifications for target systems. The concept of reuse is prevalent at several levels of the domain-modeling method and prototype environment.

The sixth article "Enveloping Sophisticated Tools into Process-Centered Environments" by Valetto and Kaiser presents a tool-integration strategy based on enveloping pre-existing tools without source-code modifications or recompilation and without assuming an extension language, application programming interface, or any other special capabilities on the part of the tool. Their strategy is intended for sophisticated tools, such as groupware applications.

Krogstie's "Use of Methods and CASE-Tools in Norway: Results from a Survey" reports the results of a survey investigation on development and maintenance representing 52 Norwegian organizations. One trend shows an increased use of packaged solutions, although larger organizations continue to develop custom applications and have in place comprehensive development and maintenance methodologies for the use of CASE tools.

The survey's results show a modest difference in the perception of CASE benefits between users and nonusers, but this result is not statistically significant.

The final article in this special issue of JASE, "A Debugging and Testing Tool for Supporting Evolutionary Software Development" by Abramson and Sosic, describes a tool for debugging programs that have been developed over long periods of time. Their tool enhances the traditional debugging approach by automating the comparison of data structures between two running programs—a program from an older generation that is known to operate correctly and a newer version that needs to be debugged. A visualization system allows the user to view the differences between the standard data structure and the revised one. The authors demonstrate the use of their tool on a small test case.

As the end of the twentieth century approaches, the CASE community of researchers, vendors, and practitioners realizes that much has been accomplished, even though our target keeps moving towards a software development environment that is more and more sophisticated and automated. Software developers around the world are searching for more sophisticated, integrated, and complete CASE environments to satisfy their ever-increasing demand for high-quality software that is delivered on time. We hope the articles in this special issue contribute positively to your search for new ideas for your software development. Best wishes for your continued success.

Automated Software Engineering, 3, 193–218 (1996)
© 1996 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Automating the Software Inspection Process

FRASER MACDONALD, JAMES MILLER, ANDREW BROOKS, MARC ROPER, MURRAY WOOD

fraser@cs.strath.ac.uk

Empirical Foundations of Computer Science (EFoCS)
Department of Computer Science, University of Strathclyde, Glasgow, U.K., G1 1XH

Abstract. Inspection is widely believed to be the most cost-effective method for detecting defects in documents produced during the software development lifecycle. However, it is by its very nature a labour intensive process. This has led to work on computer support for the process which should increase the efficiency and effectiveness beyond what is currently possible with a solely manual process. In this paper, we first of all describe current approaches to automation of the inspection process. There are four main areas of inspection which have been the target for computer support: document handling, individual preparation, meeting support and metrics collection. We then describe five tools which have been developed to support the inspection process and compare the capabilities of these tools. This is followed by a fuller discussion of the features which could be provided by computer support for inspection and the gains that may be achieved by using such support.

Keywords: Software inspection, CASE, collaborative work

1. Introduction

The inspection process was first described by Michael Fagan in 1976 (Fagan, 1976). It is a rigorous method for statically verifying documents. A team consisting of the author of the document, a moderator, a recorder and a number of inspectors proceed to inspect the document using a multi-stage process. The inspection starts with a period of planning, where the participants are selected and materials prepared. The next stage is the overview, where the group receive a briefing on the document under inspection. During preparation, each member of the team individually becomes familiar with the material. There is some debate over whether defects should be detected during this phase. Fagan (1976) states that this should be left to the next stage, while others such as Gilb and Graham (1993) advocate that many defects can be found at this point. The preparation stage is followed by the actual inspection meeting, involving the entire team. At this point the team categorise each defect for severity and type and record it for the author to fix. This meeting is followed by a period of rework, where the author addresses each defect. Finally, a follow-up is carried out to ensure each defect has been addressed.

The benefits of inspection are generally accepted, with success stories regularly published. In addition to Fagan's papers describing his experiences (Fagan 1976, 1986), there are many other favourable reports. For example Doolan (1992) reports a 30 times return on investment for every hour devoted to inspection. Russell (1991) reports a similar return of 33 hours of maintenance saved for every hour of inspection invested. This benefit is derived from applying inspection early in the lifecycle. By inspecting products as early as possible, major defects will be caught sooner and will not be propagated through to the final product, where the cost of removal is far greater.

Despite the benefits, inspection has been found to be difficult to put into practice. This can be attributed to several factors. Firstly, it requires an investment in time and money to introduce it. Although the investment is reasonable when compared with the benefits, there may be a reluctance to devote the necessary resources, especially during a project where progress has fallen behind schedule. Another factor, according to Russell (1991), is the "low-tech" image of inspection, which is contrary to today's technology saturated development environments. Russell also points out the confusion between inspection, reviews and walkthroughs. The main difference is that inspection is highly formal. Walkthroughs tend to be used for training purposes, while reviews are aimed at achieving consensus on the content of a document. Both techniques will find defects, but neither are as effective as inspection (Gilb and Graham, 1993). If a development team is already using one of these method, it may be difficult to persuade them that inspection is better.

When inspection is implemented properly, the results achieved are worthwhile, as the inspection process provides an increase both in overall product quality and in productivity (Ackerman et al., 1989). However, manual inspection is labour intensive, requiring the participation of four or more people over a long period of time. By automating some parts of the process and providing computer support for others, the inspection process has the capability of being made more effective and efficient, thus potentially providing even greater benefits than are otherwise achieved. In addition, one desirable attribute of inspection is rigour. Using computers to support the process can help provide this rigour, and improve the repeatability of the inspection process. Repeatability is essential if feedback from the process is to be used to improve it.

In Section 2 we describe current approaches to tool support for inspection. Section 3 describes currently available tool support and in Section 4 we evaluate that tool support. In Section 5, we describe the features we believe an inspection support tool could provide, along with the ways in which the process may be thus improved. Section 6 concludes the paper.

2. Current Approaches to Automating the Inspection Process

In this section we describe the features of inspection which have been tackled by current inspection tools. These features fall under four broad categories: document handling, individual preparation, meeting support and data collection.

Document Handling The most obvious area for tool support is document handling. Traditional inspection requires the distribution of multiple copies of each document required. Apart from the cost and environmental factors associated with such large amounts of paper, cross-referencing from one document to another can be very difficult. Since most inspection documents are produced on computer, it is natural to allow browsing of documents on-line. Everyone has access to the latest version of each document, and can cross-reference documents using, for example, hypertext. These features demonstrate that computerising documents is not simply a change of medium, but provides an opportunity to enhance the presentation and usability of those documents.

The comments produced by inspectors indicate when and where an inspector takes issue with the document. In the traditional inspection, they are recorded on paper. Computer support allows them to be stored on-line, linked to the part of the document to which they refer. They can then be made available for all inspectors to study before and during the inspection meeting. This has the added advantage of helping to reduce the inaccuracies and mistakes which can occur during the inspection meeting, including the failure to record some comments altogether. This effect has been observed by Votta (Votta, 1993) and can occur in several situations, including when inspectors are unsure of the relevance of their comments. By storing all comments on-line, it is easier to ensure that each one is addressed.

Individual Preparation There are several ways in which tool support can assist in individual preparation, in addition to the document handling and annotation facilities described above. Automated defect detection can be used to find simple defects such as layout violations. This type of defect, while not being as important as such items as logic defects, must still be found to produce a correct document. If finding them can be automated, inspectors can concentrate on the more difficult defects that cannot be automatically found and that have a potentially greater impact if not found. This may be achieved by the introduction of new tools, or the integration of the inspection environment with existing tools. The latter is obviously preferable. There are various levels of integration, from simply reporting defects to actually producing an annotation relating to the defect for the reviewer to examine.

Computer support can provide further help during individual preparation. Generally, inspectors make use of checklists and other supporting documentation during this stage. By keeping these on-line, the inspector can easily cross-reference between them. On-line checklists can also be used by the tool to ensure that each check has been applied to the document, thereby enforcing a more rigorous inspection, while on-line standards, such as those pertaining to the layout of documents, assist the inspector in checking a document feature for compliance.

Meeting Support Intentionally, or otherwise, some members of the team may not spend sufficient time on individual preparation, but will still attend the group meeting and try to cover up their lack of preparation. Inevitably, this means that the inspector in question will have little to contribute to the group meeting, thus wasting both the group's time and the inspector's time. Computer support can help avoid this situation by monitoring the amount of time spent by each inspector in preparation. The moderator can use this information to exclude anyone who has not prepared sufficiently for the group meeting, or to encourage them to invest more effort. The moderator can also decide when is the best time to move from the preparation stage to the meeting, taking account of the amount of preparation performed by each inspector.

Since guidelines state that a meeting should last for a maximum of only two hours (Fagan, 1976), it may take many meetings to complete an inspection. There is a large overhead involved in setting up each meeting, including finding a mutually agreeable time, a room to hold the meeting and so forth. There is also an overhead involved for each participant travelling to the meeting. By allowing a distributed meeting to be held using conferencing technology, it may be easier for team members to 'attend' the meeting using any suitably equipped workstation.

An alternative solution to the meeting problem is to remove the meeting stage altogether, performing the inspection *asynchronously*. In this type of inspection, each inspector can perform their role independently. The inspection moves from stage to stage when every inspector has completed the required task. This type of inspection can also reduce the meeting losses referred to before.

When a meeting is taking place, it can sometimes be useful to conduct polls to quickly resolve the status of an issue. This is especially important if the meeting is being held in a distributed environment. Computer support can allow polls to be quickly taken, thus helping the inspection meeting progress rapidly.

Data Collection An important part of inspection is the collection of metrics which can be used to provide feedback to improve the inspection process. The metrics will include such data as time spent in meeting, defects found, overall time spent in inspection and so forth. Collecting these metrics is time-consuming and error-prone when carried out manually, so much so that Weller (1993) states:

"...you may have to sacrifice some data accuracy to make data collection easier..."

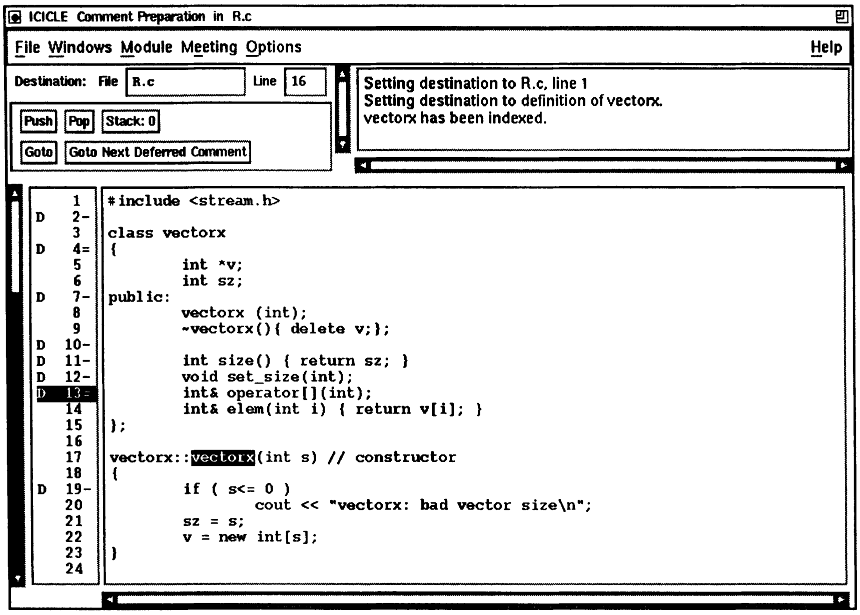
This is obviously undesirable. Computer support allows metrics from the inspection to be automatically gathered for analysis. This removes the burden of these dull but necessary tasks from the inspectors themselves, allowing them to concentrate on the real work of finding defects. Furthermore, the computer can often be used for analysing these metrics with little further work. This is unlike manual data collection, where the data has to be entered before it can be analysed. Automated data collection also has the advantage of being less error-prone than its manual counterpart.

3. Current Support for Automated Inspection

In this section we describe currently available tool support for inspection in terms of the areas described in the previous section. Although all tools described have the aim of improving the inspection process, each has its own approach. Additionally, some of the tools use variations of the Fagan inspection process. The variation used will be described along with the tool which supports it.

3.1. ICICLE

ICICLE (Intelligent Code Inspection in a C Language Environment) (Brothers et al., 1990; Sembugamoorthy and Brothers, 1990), as its name suggests, is an automated intelligent inspection assistant developed to support the inspection of C and C++ code. This inspection tool is unique in making use of knowledge to assist in finding common defects. Since the knowledge is of a very specific kind, ICICLE is less suitable for supporting general inspection. It can, however, be used to inspect plain text files by turning off the initial analysis. The tool is designed to support two phases of inspection: comment preparation (individual) and the inspection meeting itself. During the inspection meeting, the tool provides the functionality available in individual checking, supplemented by support for cooperative working.



```
Setting destination to R.c, line 1
Setting destination to definition of vectorx.
vectorx has been indexed.
```

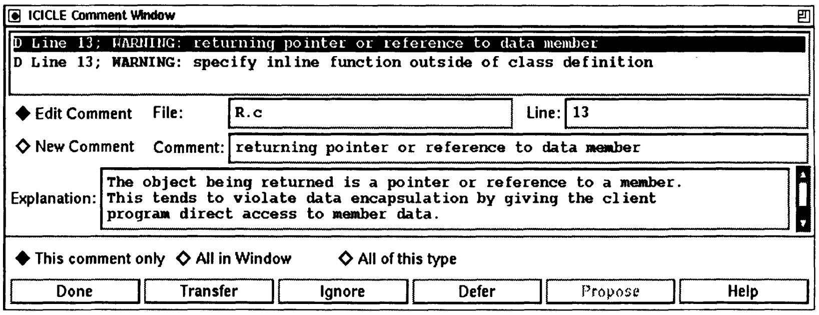
```
    1  #include <stream.h>
D   2-
    3  class vectorx
D   4= {
    5          int *v;
    6          int sz;
D   7- public:
    8          vectorx (int);
    9          ~vectorx(){ delete v;};
D  10-
D  11-         int size() { return sz; }
D  12-         void set_size(int);
D  13=         int& operator[](int);
   14          int& elem(int i) { return v[i]; }
   15  };
   16
   17  vectorx::vectorx(int s) // constructor
   18  {
D  19-         if ( s<= 0 )
   20                  cout << "vectorx: bad vector size\n";
   21          sz = s;
   22          v = new int[s];
   23  }
   24
```

Figure 1. The main ICICLE display.

Document Handling The source code is displayed in a large window with each line numbered (see Figure 1). This window can be augmented by a second code window, allowing the user to compare two parts of the code simultaneously. Next to the line numbers are two symbols referring to comments. A letter indicates the status of the comment. This can include *deferred* (not dealt with yet), *ignored* (user decides the comment is inappropriate or otherwise suspect) or *transferred* (chosen to be discussed at the inspection meeting). The second symbol indicates the presence of a comment for this line. A hyphen indicates a single comment, while an equals represents multiple comments.

By clicking on the appropriate line, a comment window for that line is raised. A typical comment window is shown in Figure 2. This window allows a comment to be modified or inserted and its status changed. Any changes to this comment can be propagated to all comments on the line or even all commments in the code which have the same text.

Individual Preparation ICICLE can automatically prepare comments on source code using its analysis tools. These include the UNIX tool `lint` and ICICLE's own rule-based static debugging system. `lint` can be used to detect certain defects in C code, such as unreachable statements and possible type clashes. The ICICLE rule-based system can be used to flag both serious defects, such as failure to deallocate memory, and more minor defects, such as standards violations. There is also the ability to include customised analysis procedures. The comments produced by all these tools can either be accepted by the inspectors if they agree with them, modified or else completely rejected. ICICLE also provides a facility



ICICLE Comment Window

```
D Line 13; WARNING: returning pointer or reference to data member
D Line 13; WARNING: specify inline function outside of class definition
```

◆ Edit Comment File: R.c Line: 13

◇ New Comment Comment: returning pointer or reference to data member

Explanation: The object being returned is a pointer or reference to a member. This tends to violate data encapsulation by giving the client program direct access to member data.

◆ This comment only ◇ All in Window ◇ All of this type

Figure 2. The ICICLE comment preparation window.

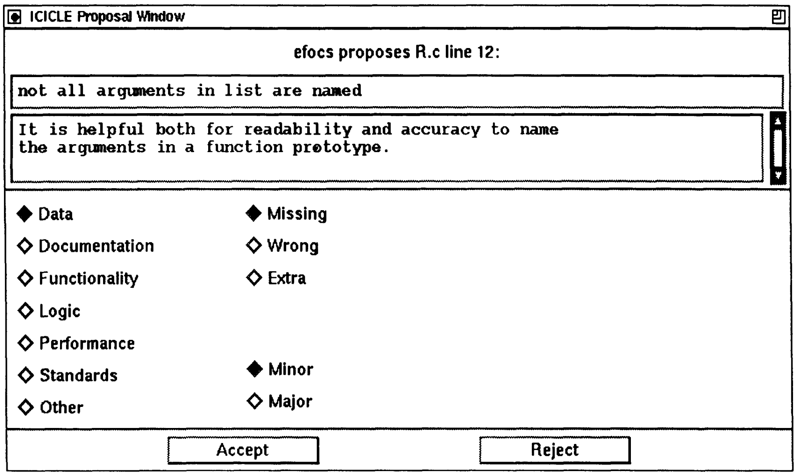
to allow browsing of Unix manual pages. The system also provides cross referencing information for 'objects' such as variables and functions. For example, clicking on the use of a variable would give the user an option to move to the point of declaration, or any other usage of the variable. This facility is available over multiple source files.

Meeting Support The inspection meeting is held with every inspector using ICICLE in the same room. No provision is made for distributed meetings, since it is felt by the authors that they "do not wish to supplant the ordinary verbal medium by which the bulk of meeting communication occurs" (Sembugamoorthy and Brothers, 1990).

During the inspection meeting, each inspector has access to all documents as well as their own comments. Each inspector has the code window displayed on screen. The reader controls the traversal of this window for all participants, just as a single inspector does during comment preparation. Every code window is locked to the reader's view, although an inspector can open an extra window to allow simultaneous inspection of two sections of the code.

The reader proceeds through the document until an issue is proposed by an inspector. When this happens, a proposal window appears on all displays. The scribe's proposal window is shown in Figure 3. The inspection team discuss the comment, and when discussion is complete, the scribe is able to classify the comment and accept it, or reject the comment completely. If the comment is accepted it is stored in a file which becomes the output of the meeting. The other participants windows are similar, but lack the controls for classifying a comment. During the meeting, participants can send single line text messages to all other participants.

Data Collection When the inspection meeting is complete, ICICLE generates a list of all accepted defects to be given to the author of the product under inspection. A summary of the defects by type, class and severity is also generated. The scribe can also prepare a report detailing the total time spent in preparation and in meeting, the inspectors present and other such process information.


ICICLE Proposal Window

efocs proposes R.c line 12:

```
not all arguments in list are named
```

```
It is helpful both for readability and accuracy to name
the arguments in a function prototype.
```

◆ Data	◆ Missing
◇ Documentation	◇ Wrong
◇ Functionality	◇ Extra
◇ Logic	
◇ Performance	
◇ Standards	◆ Minor
◇ Other	◇ Major

Figure 3. The ICICLE comment proposal window.

3.2. Collaborative Software Inspection

Collaborative Software Inspection (CSI) (Mashayekhi et al., 1993), is designed to support inspection of all software development products. The tool is described as applied to the Humphrey model of inspection (Humphrey, 1989). In this variation, each inspector creates a list of faults during individual inspection, which are then given to the author of the document before the inspection meeting. It is the author's task to correlate these fault lists and to then address each defect at the inspection meeting.

Document Handling CSI provides a *browser* for viewing the document under inspection, which automatically numbers each line. When a line is selected, an *annotation* window pops up, allowing the inspector to make a comment about that particular line. This annotation is supported by hyperlinks between the annotation itself and the document position to which it refers. Since annotations can only refer to one line, and there may be a need for general comments about an area of the document, CSI also supports a *notepad* system, which allows annotations about missing material.

Individual Preparation Support is available from CSI for detecting defects by provision of on-line *criteria* which help the inspector determine faults. Also, when recording annotations, the inspector is given guidance in categorising and sorting faults. After all inspectors have finished individual inspection, the author can access all annotations associated with the document and correlate them into a single *fault list*, supported by CSI through automatically summarising and integrating the individual fault lists. The author can then categorise each fault, either accepting it or rejecting it. CSI also allows the author to sort the fault list on multiple keys, including severity, time of creation and disposition.

Meeting Support At the inspection meeting, the document under inspection is made visible on a window on each inspector's screen. The author guides the meeting using the correlated fault list. Each item is discussed, and when agreement is reached regarding its severity, this is noted by the recorder in the *action list*. Note that the original annotations are available at this point to help inspectors understand the nature of the fault, and further annotations can be added during the meeting. When the end of the fault list is reached, the inspectors agree on the status of the meeting, indicating whether the material under inspection is to be accepted or reinspected. This is noted by the recorder. CSI provides for distributed inspection, allowing an inspection meeting to be carried out with team members in a variety of disparate locations. This is supported by a teleconferencing tool called Teleconf (Reidl et al., 1993), which provides audio for the meeting.

The developers of CSI have moved on to a new prototype inspection system (Mashayekhi et al. 1994). Called CAIS (Collaborative Asynchronous Inspection of Software), the system uses the CSI system for annotating documents. However, the system is designed to be used asynchronously and does not rely on having all inspection participants present for any part of the process. This asynchrony can reduce the amount of time required to complete the inspection, since there is no need to find a common time when all inspectors are free to carry out the meeting.

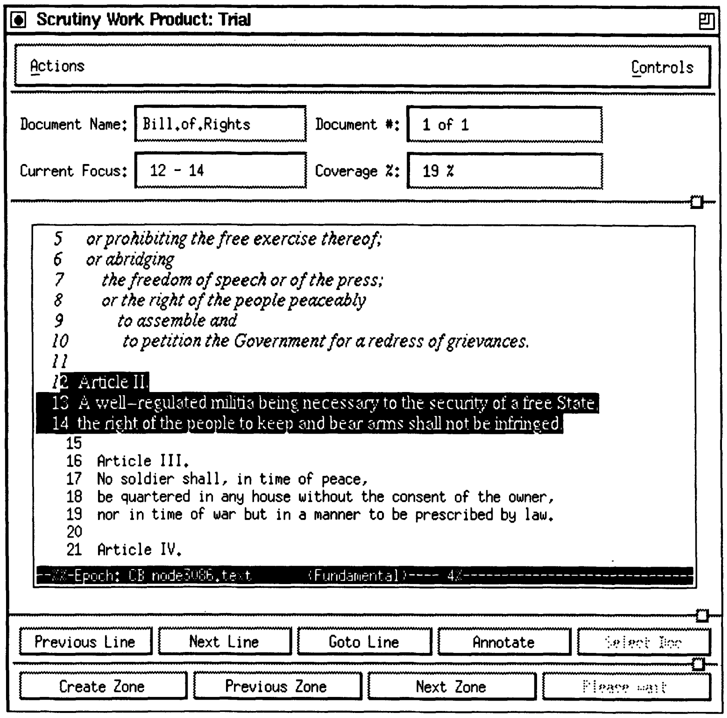
Data Collection The *inspection summary* is used to record meeting information such as team members present, their roles and the status of the inspection meeting. CSI also provides a *history log*. This collects several metrics from the process, such as the time spent in the meeting and the time taken to find a fault, as well as the number and severity of faults found.

3.3. Scrutiny

Scrutiny (Gintell et al., 1993) is an inspection tool based on the inspection method used at Bull HN Information Systems. This process consists of four stages. The first stage is *initiation* and is comparable to overview in the Fagan model. The second stage is *preparation*, as in the Fagan model. The inspection meeting itself is called *resolution*, while the final stage, *completion*, encompasses both rework and follow-up. The roles taken by each participant are also similar, however Scrutiny also implements some changes. First, the moderator's role is changed to include the duties of the reader. In addition, the recorder role can be taken by more than one person. Scrutiny also explicitly implements the role of the producer, who can answer questions regarding the document. Finally, there is another role in the form of the *verifier* who ensures the defects found by the inspection team have been correctly addressed by the author. This role may be assigned to any participant. Any other members of the team are cast as inspectors. Each stage of the process, along with each of the four roles, is modelled in Scrutiny.

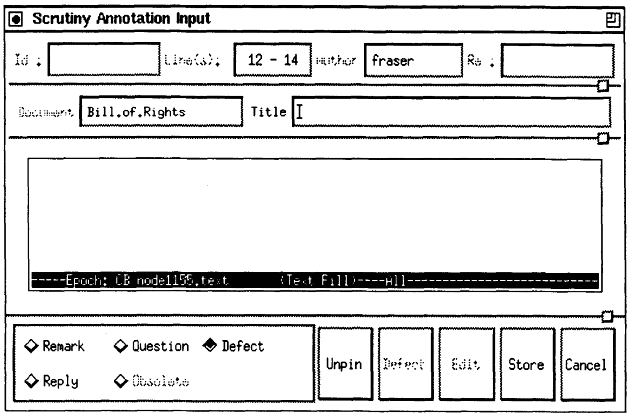
Document Handling The *work product window* allows each inspector to view the document under inspection (see Figure 4). The document is displayed with each line numbered and the current focus indicated by reverse video. The current focus is usually a single line but may also be a zone of several lines. Text which has been inspected is italicised, and the percentage of the document covered is displayed in the top right hand corner. The window has controls to move through the document line by line, and also has controls to mark a zone. Finally, there is a button to enable the creation of a new annotation.

When an annotation is created or modified, it appears in an *annotation window*, an example of which is given in Figure 5. This displays the line numbers to which the annotation refers



```
Scrutiny Work Product: Trial
Actions                                                   Controls

Document Name: Bill.of.Rights     Document #: 1 of 1
Current Focus: 12 - 14            Coverage %: 19 %

 5  or prohibiting the free exercise thereof;
 6  or abridging
 7    the freedom of speech or of the press;
 8    or the right of the people peaceably
 9      to assemble and
10      to petition the Government for a redress of grievances.
11
12  Article II.
13  A well-regulated militia being necessary to the security of a free State,
14  the right of the people to keep and bear arms shall not be infringed.
15
16  Article III.
17  No soldier shall, in time of peace,
18  be quartered in any house without the consent of the owner,
19  nor in time of war but in a manner to be prescribed by law.
20
21  Article IV.
--%%-Epoch: CB node3086.text        (Fundamental)---- 4%---------------------------
```

Figure 4. The Scrutiny work product window.



```
Scrutiny Annotation Input
Id :          Line(s): 12 - 14   Author fraser      Re :
Document Bill.of.Rights   Title I

-----Epoch: CB node1155.text        (Text Fill)----All------------------------
```

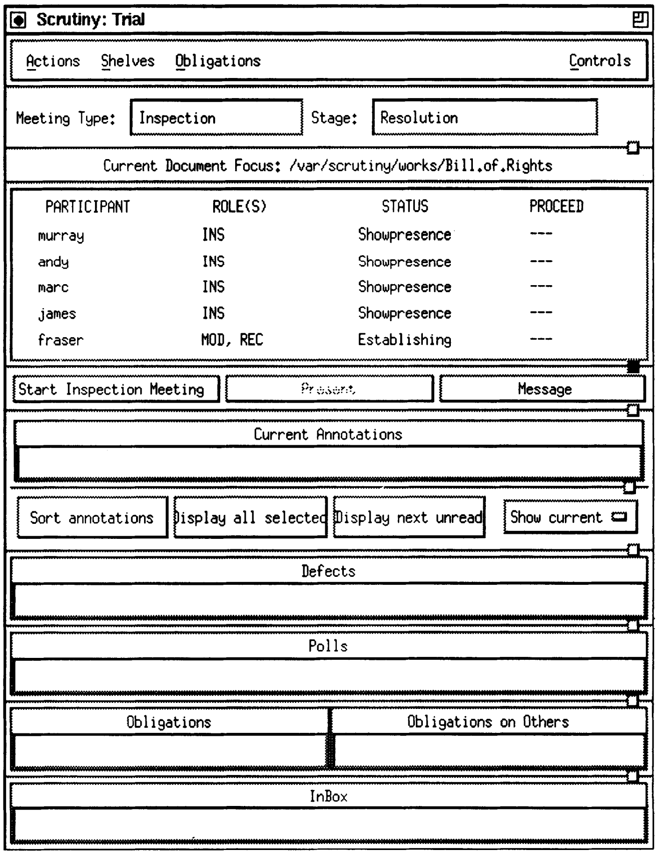
Figure 5. The Scrutiny annotation window.

and the author of the annotation, along with its content and a title. Buttons allow the type of annotation to be recorded as either a question, potential defect, remark or reply. When an annotation is created, an icon appears beside the line or zone to which it refers. Scrutiny currently only supports text documents. It is hoped to overcome this by integrating it with other tools.

Individual Preparation Here, Scrutiny simply allows the inspector to traverse the document, making annotations which can be used during the resolution stage. There is no assistance with checklists or other supporting documentation. All defects must be found manually.

Meeting Support Before the inspection meeting is started, the moderator can view the preparation time of each inspector, to ensure that enough time has been given to allow adequate preparation. Each inspector also has the opportunity to add time for any off-line preparation which they may have engaged in.

During the meeting, the work product window is used by each participant to view the document, with the moderator having additional controls to change the current focus and to initiate a poll. The moderator guides the inspectors through the document, while they read and discuss the annotations made. Polls are used to resolve the status of an annotation.



Meeting Type: Inspection Stage: Resolution

Current Document Focus: /var/scrutiny/works/Bill.of.Rights

PARTICIPANT	ROLE(S)	STATUS	PROCEED
murray	INS	Showpresence	---
andy	INS	Showpresence	---
marc	INS	Showpresence	---
james	INS	Showpresence	---
fraser	MOD, REC	Establishing	---

Figure 6. The Scrutiny control window.

Scrutiny also provides a main control panel called the *control window*, a copy of which is seen by each inspector (Figure 6). This window consists of four major subwindows. The *participant status* display contains a list of the participants along with an indication of their current activities. The *annotations* subwindow contains a list of annotations made on the current document, along with their owners and a type. The *defect* subwindow lists defect reports that have been discussed and their status agreed upon. The status includes the type and severity of the defect. The final subwindow is concerned with *polls*. Every time a poll is taken during the inspection meeting to resolve an issue, a record of it is kept here.

Scrutiny can be used for both same-place and distributed inspection. The latter makes use of both audio and teleconferencing facilities. It is also possible to hold distributed inspections without these audio and teleconferencing facilities by making use of Scrutiny's built in textual communications systems. The discussion client allows inspectors to exchange textual points of discussion. Each participant has a list of the current discussion points which can be read and replied to. Replies have a reference to the original point, and participants can traverse these chains of points, allowing them to follow a discussion and then add their own comments. Scrutiny also provides a means of sending a simple message to meeting participants. In addition to composing your own message, there are several frequently required messages, such as a request to move to the previous line, which can quickly be sent. These messages can be sent to named individuals, or the group as a whole. It is not clear how effective these mechanisms are when holding a synchronous meeting, since the medium is obviously not as information rich as face-to-face communication, even that provided by teleconferencing.

Data Collection Scrutiny automatically generates an inspection report containing all the relevant information about the inspection and its participants, details of the time spent by each participant in the inspection and the coverage of the document they achieved. It also contains a complete defect list with summary information.

3.4. InspeQ

InspeQ (Inspecting software in phases to ensure Quality) is a toolset developed by Knight and Myers (1991; 1993) to support their phased inspection technique. The technique was developed by Knight and Myers with the goal of permitting the inspection process to be "rigorous, tailorable, efficient in its use of resources, and heavily computer supported" (Knight and Meyers, 1991).

A phased inspection consists of an ordered set of phases, each of which is designed to ensure the product possesses either a single, specific property or a small set of related properties. The phases are ordered so that each phase can build on the assumption that the product contains properties that were inspected for in previous phases. The properties that can be checked for are not necessarily those concerned purely with defects of functionality. They can include such qualities as reusability, portability and compliance with coding standards.

There are two types of phase: single-inspector and multiple-inspector. A single-inspector phase uses a rigorous checklist. The product either does or does not comply with each item on the checklist. The phase cannot be completed until the product satisfies all checks.

These phases are carried out by a lone inspector. Multiple-inspector phases are designed for properties which cannot easily be described by binary questions. The product is first examined individually by each inspector. This individual checking makes use of a checklist that is both application specific and domain specific, though the questions are not binary, as they are in the single-inspector phase. The individual checking is followed by a meeting, called a *reconciliation*, in which the inspectors compare their findings. In a good inspection, these results from all inspectors will be very similar. Note that although it is not designed to do so, the reconciliation provides a further opportunity for fault detection.

Phased inspections are designed to allow experts to concentrate on finding defects that they have specialised knowledge of, thus making more efficient use of human resources. For example, it may be more efficient to have domain analysts inspecting code for reusability, since they will have expert knowledge in that particular field.

Document Handling The *work product display* is the major tool used during inspection, allowing the inspector to browse the document under inspection. By using multiple copies, the inspector can simultaneously examine separate parts of the same document. It also allows the inspector to search the document. The *comments display* allows the inspector to note any issues found. To provide context for each issue, parts of the text or line numbers can be pasted into the comments window. This can also be used to demonstrate a defect by example: the inspector pastes in the incorrect version and then suggests a correct version. InspeQ carries out formatting of these comments before they are passed on to the author.

Individual Preparation A *checklist display* is used to display the checklist associated with the current inspection. This ensures that the inspector knows exactly what is required to be examined in this phase. The checklist also allows the inspector to indicate completion of each check, by marking each item as *complies*, *does not comply*, *not checked* or *not applicable*. To help enforce a rigorous inspection, InspeQ ensures that all checklist items are addressed by the inspector before the product exits the phase. A future extension will ensure that each checklist item is applied to every feature associated with that item. Checklists usually ensure compliance with one or more standards. To help the inspector apply the checklist, a *standards display* is available which presents each standard in full, as well as providing examples.

The *highlights display* can allow the inspector to quickly identify specific features of the document. These can be highlighted but can also be displayed in a separate window for examination. An example would be to highlight all the `while` statements in a C program to allow them to be checked for correctness, without the distraction of the surrounding code. This function requires syntactic information about the document, which is more readily available for code than any other type of document.

Meeting Support Since InspeQ is designed for individual inspector use, there is no support for group meetings.

Data Collection Again, InspeQ is designed for individual inspector use, and only generates the comment list for each inspector. These lists are then compared at the reconciliation.

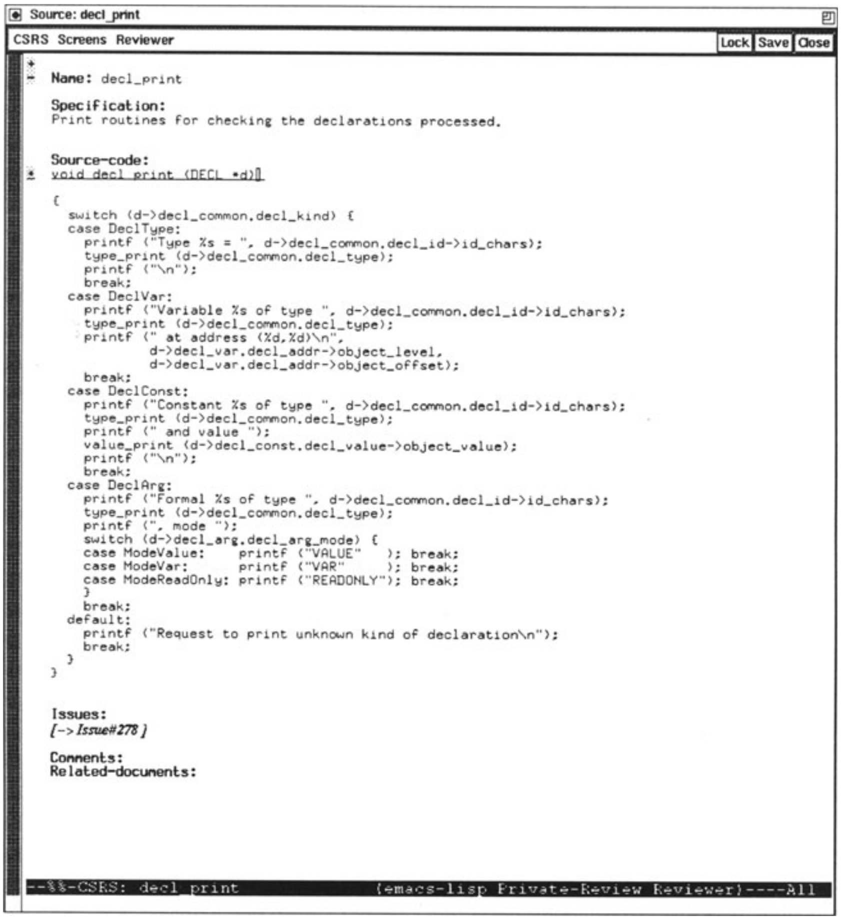
3.5. Collaborative Software Review System

Collaborative Software Review System (CSRS) (Johnson, 1994a) is an environment to support the use of FTArm (Formal Technical Asynchronous review method) (Johnson and Tjahjono, 1993), a development of Fagan inspection. FTArm is a general method for inspecting any type of document, consisting of six phases. The first is *Setup*, which involves choosing the members of the inspection team and preparing the document for inspection via CSRS. This involves organising the document into a hypertext structure and entering it into the database. The document is held as a series of linked nodes, with each node containing some feature of the document. In the case of source code, each node would be a function, variable or similar item. *Orientation* is equivalent to Overview in the Fagan process, and may involve a presentation by the author. The goal is to familiarise the team with the work under inspection. *Private Review* is similar to Preparation. The inspector reads each source node in turn, and has the ability to create new nodes containing annotations. When each reviewer has covered each node (or sooner, if required), the inspection moves on to the next phase. In *Public Review*, all nodes become public and inspectors can asynchronously vote on the status of each one, either confirm, disconfirm or neutral. Additional nodes can be created at this stage, immediately becoming public. When all nodes have been resolved, or if the moderator decides that further voting and on-line discussion will not be fruitful, the public phase is declared complete. During *Consolidation* the moderator writes a report detailing the results from the private and public review phases, including summarised comments of inspectors. The moderator also decides whether a meeting is to be held to resolve any remaining issues. If not, the report is distributed for signature by each reviewer. The final phase is the *Group Review Meeting* which is used to solve any unresolved issues remaining from the private and public review phases. The final inspection report is then produced by the moderator.

As can be seen from the above description, FTArm is fundamentally different to the traditional inspection process. Instead of consisting of an asynchronous and a synchronous phase, almost the entire inspection is held asynchronously. This has great advantages in making the inspection more flexible, since there is much less need for everyone to be in the same place at the same time, but the effectiveness of such a technique has not yet been empirically evaluated.

CSRS is probably the most flexible of all tools described here as it can be customised to support different variants of the inspection process. This is accomplished using a process modelling language (Johnson, 1994b). This language has several facilities, including constructs for defining phases of the method, a construct for defining the role of each participant, and constructs to define the artifacts used during the inspection. The latter also includes support for checklists. The language can also be used to define the user interface, as well as to control the type of data analysis carried out by CSRS.

Document Handling A document is stored in a database as a series of nodes. For source code, these nodes would consist of functions and other program constructs. Source nodes are created at the start of the inspection by the document author with the aid of the moderator. The nodes are connected via hypertext-style links, allowing the inspector to traverse the document. A typical source node is displayed in Figure 7. The name of the function is

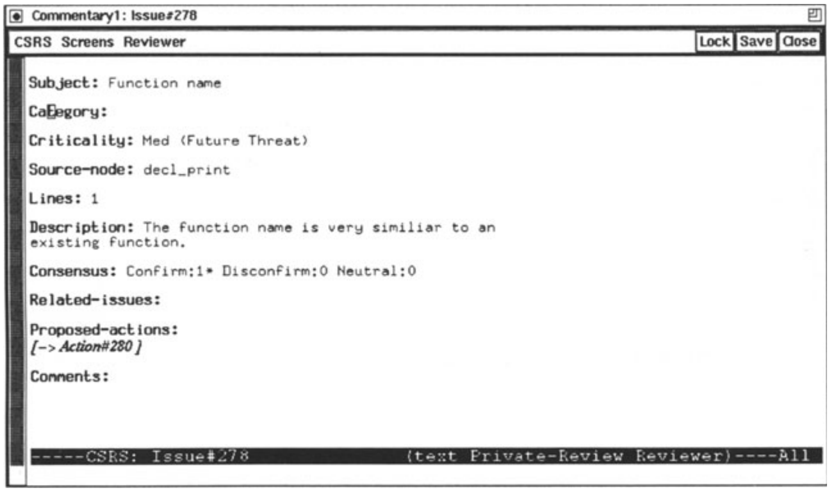


```
Name: decl_print

Specification:
Print routines for checking the declarations processed.

Source-code:
void decl_print (DECL *d)

{
  switch (d->decl_common.decl_kind) {
  case DeclType:
    printf ("Type %s = ", d->decl_common.decl_id->id_chars);
    type_print (d->decl_common.decl_type);
    printf ("\n");
    break;
  case DeclVar:
    printf ("Variable %s of type ", d->decl_common.decl_id->id_chars);
    type_print (d->decl_common.decl_type);
    printf (" at address (%d,%d)\n",
            d->decl_var.decl_addr->object_level,
            d->decl_var.decl_addr->object_offset);
    break;
  case DeclConst:
    printf ("Constant %s of type ", d->decl_common.decl_id->id_chars);
    type_print (d->decl_common.decl_type);
    printf (" and value ");
    value_print (d->decl_const.decl_value->object_value);
    printf ("\n");
    break;
  case DeclArg:
    printf ("Formal %s of type ", d->decl_common.decl_id->id_chars);
    type_print (d->decl_common.decl_type);
    printf (", mode ");
    switch (d->decl_arg.decl_arg_mode) {
    case ModeValue:    printf ("VALUE"   ); break;
    case ModeVar:      printf ("VAR"     ); break;
    case ModeReadOnly: printf ("READONLY"); break;
    }
    break;
  default:
    printf ("Request to print unknown kind of declaration\n");
    break;
  }
}


Issues:
[->Issue#278 ]

Comments:
Related-documents:

--%%-CSRS: decl print              (emacs-lisp Private-Review Reviewer)----All
```

Figure 7. The main CSRS window.

given, followed by a specification of its intended function. This is followed by the source code itself.

Annotations are also stored as nodes, and can be one of three types. The first type is a *Comment* node, which is used to raise questions about the document and to answer them. These are made public to all inspectors. An *Issue* node indicates a perceived defect in the source node. Issue nodes are initially private to individual reviewers. An example issue node is given in Figure 8. This issue is linked to the source code in Figure 7, where a link to the issue node can be seen near the bottom of the display. Finally, an *Action* node is a suggestion of the action that should be taken to resolve an issue. These are also private to reviewers. The action node given in Figure 9 details a possible fix for the issue raised previously.



```
Subject: Function name
Category:
Criticality: Med (Future Threat)
Source-node: decl_print
Lines: 1
Description: The function name is very similiar to an
existing function.
Consensus: Confirm:1* Disconfirm:0 Neutral:0
Related-issues:
Proposed-actions:
[-> Action#280 ]
Comments:

-----CSRS: Issue#278            (text Private-Review Reviewer)----All
```

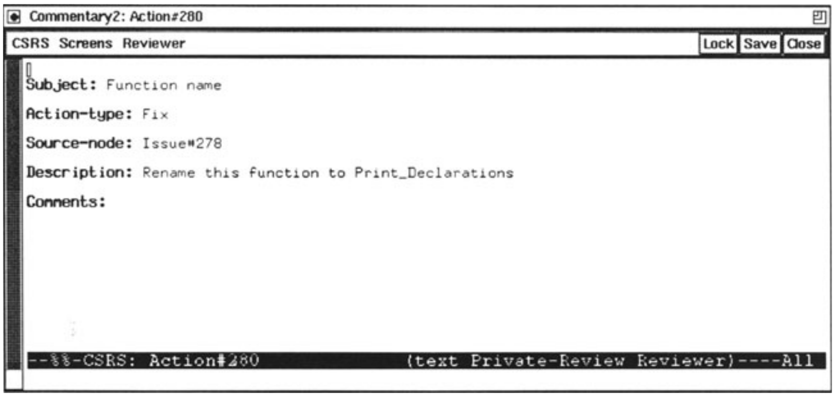
Figure 8. A CSRS issue node.

Individual Preparation The FTArm method predominantly consists of individual work, and this is where CSRS provides the most support. During the private review phase, each inspector has a summary of which nodes have been covered and which have still to be covered. This information is also available to the moderator, who will use it to decide when to move on from private review to the next phase. Since additional nodes may be created after a reviewer has reviewed all the currently available nodes, CSRS has the facility to automatically e-mail all reviewers when new nodes are created and have to be reviewed. CSRS also provides an on-line checklist of standard issue types to assist the reviewer.

Support during public review is similar to that for private review, except now all nodes are publicly accessible. This time the main focus is on issue nodes. Each reviewer has to visit each node, where CSRS can be used to vote on that node's status. Again, the reviewer has summary information available, indicating which nodes have still to be visited. The moderator can also use this information to decide when to terminate public review, usually when all reviewers have visited all nodes.

Meeting Support CSRS has little in the way of group meeting support, due to the predominantly asynchronous nature of the inspection method implemented. The group review meeting must be held face-to-face in the traditional manner. CSRS does not provide any support except to help the moderator summarise the results and produce a LaTeX formatted report.

Data Collection CSRS provides automatic collection of such data as number and severity of defects and time spent reviewing each node. It also has the ability to keep an event log, which details the entire inspection from start to finish, allowing detailed (manual) analysis later on.



```
Subject: Function name
Action-type: Fix
Source-node: Issue#278
Description: Rename this function to Print_Declarations
Comments:

--%%-CSRS: Action#280          (text Private-Review Reviewer)----All
```

Figure 9. A CSRS action node.

Table 1. Summary of features of currently available inspection tools.

	ICICLE	CSI	Scrutiny	InspeQ	CSRS
Text	●	●	●	●	●
Linked Annotations	●	●	●		●
Cross-referencing	●				
Automated Analysis	●				
Checklists		●		●	●
Supporting Documentation	●			●	
Enforcement			●	●	●
Distributed Support		●	●	(individual)	
Decision Support			●		●
Metrics Collection	●	●	●		●

4. Comparison of Existing Tools

Table 1 summarises the features of each tool. It can be seen that while basic document inspection and annotation is well-supported, the more advanced features described in Section 2 are less common. Here we compare the features supported by each tool.

Document support All the tools described handle plain text documents adequately. ICICLE, Scrutiny and CSI use the same technique of displaying the document with each line numbered. Annotations can then be made which are linked to an individual line. Scrutiny also uses the idea of a current focus, which is a current sentence of interest upon which an annotation can be made. CSRS divides the document up into smaller chunks called nodes, each of which can be inspected on its own and comments made via new nodes linked to this one. InspeQ is the least well supported in this area, since comments are completely separate from the source document, with only cut and paste facilities available to give a context to a comment. In essence this only gives the facilities that a good text editor can supply.

ICICLE, CSI, Scrutiny and CSRS all allow classification of annotations, while InspeQ only allows their creation or deletion. This limits the scope for collection of defect type metrics, although it still allows the overall number of defects to be measured.

Individual Preparation Checklists are supported only by InspeQ, which uses them to enforce a rigorous inspection by ensuring each item on the checklist is attended to by the inspector. In a similar vein, CSI has the concept of a criteria list which helps inspectors find and categorise faults. InspeQ also supports the displaying of standards, while ICICLE can provide a browsing facility for manual pages like those provided in Unix. CSRS has only a checklist of issue types, while Scrutiny has no support in this area at all.

ICICLE is the only tool to provide any automatic defect detection. This is currently provided using the UNIX tool `lint` and ICICLE's own rule based system, which contains knowledge about C source code that can be used to detect such defects as coding violations.

Meeting Support To ensure that each inspector has spent sufficient time in preparation, CSRS can provide details on the amount of time spent on inspection by each inspector. This prevents inspectors misleading the moderator about their state of preparation. The checklists in InspeQ also perform this function. Scrutiny stores the percentage of document covered by each inspector, as well as the time spent by each inspector in both preparation and meeting.

In terms of support for distributed meetings, CSI uses Teleconf, which provides an audio channel only. Scrutiny also supports the use of an audio channel, in addition to its discussion and messaging facilities. CSRS has no conferencing facilities since most of the inspection takes place asynchronously. ICICLE also lacks these facilities and is designed to be used when the inspection meeting takes place in one room with all inspectors present. The InspeQ toolset is designed for individual inspector use only and therefore lacks any conference facilities.

Decision support is available through polls in CSRS and Scrutiny. Neither ICICLE, CSI nor InspeQ provide such support. In the case of InspeQ this is because the toolset is not used in the group meeting at all.

Data Collection ICICLE automatically gathers metrics on the number and type of comments made, as well as their severity, as noted by the scribe during the inspection meeting. CSI uses a history log to record defect metrics including severity, time taken to find the defect and overall length of time spent in meeting. CSRS and Scrutiny have the most comprehensive metric gathering capability. CSRS has the ability to gather defect metrics, as well as fine-grained metrics on the amount of time spent by each inspector reviewing each node. Scrutiny has similar collection facilities, including the time spent in inspection and the coverage of the document achieved by each inspector. InspeQ provides no facility for metric gathering.

5. An Informal Specification of an Inspection Support Tool

The previous sections have described current approaches to automating the software inspection process and tools which implement these approaches. In this section, we discuss a more comprehensive set of features that we feel an inspection support tool could provide, along with the way in which these features may improve the inspection process.

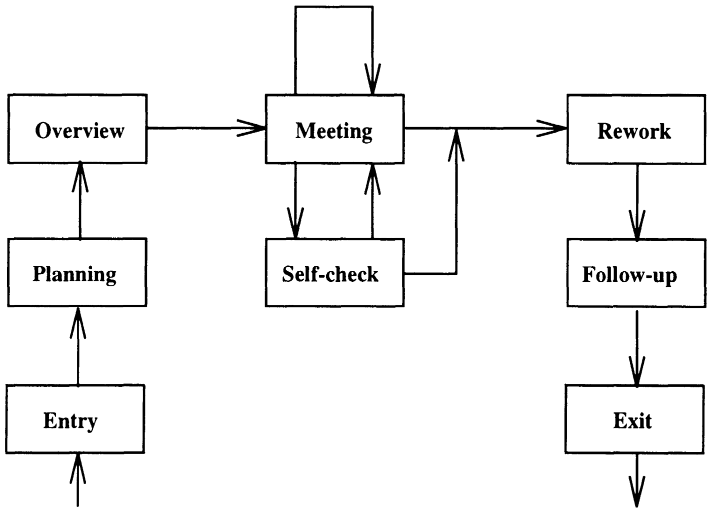


Figure 10. A model of a generic inspection process.

A model of a generic software inspection process is given in Figure 10. This model has been derived from eight well-known inspection methods, and is more fully described in Macdonald and Miller (1995). The most obvious extensions to the Fagan method described in the introduction are the entry and and exit phases, proposed by Gilb and Graham (1993), and the flexible meeting structure, which is required to model all inspection variations. Each inspection consists of a number of these meetings, each of which may have different objectives. In the following discussion we assume there are two such meetings: one for individual preparation and a group inspection meeting. The *self-check* phase in the model is intended to allow the results of a meeting to be validated. This is usually performed by the moderator and can be thought of as a single-person inspection of the results. The self-check phase will not be discussed further.

5.1. Entry and Exit

An entry phase is used to ensure that certain criteria are met *before* the inspection begins, ensuring that the inspection effort is not wasted. These criteria usually indicate that the document is ready for inspection. The exit phase involves ensuring that some criteria are met before the inspection is completed. Typical exit criteria include checking that the estimated number of defects left in the document is below an acceptable threshold and that a suitable inspection rate was adopted. This rate is the average amount of material inspected in an hour. An inspection support tool should simply present these criteria and allow them to be answered, usually by the moderator. If the criteria are satisfied, the tool can allow the inspection to begin or to complete, as appropriate.

5.2. *Planning and Overview*

An inspection is performed by people with well-defined roles and therefore those roles should be modelled by the tool. The capabilities given to each participant by the tool should match those roles. For example, only the moderator should be able to set up a meeting and add or remove participants. The tool should also be capable of supporting every stage of the inspection from start to finish. Although inspection is a well-defined process, which can be rigorously enforced by automation, the inspection tool should also be flexible enough to allow the process to be tailored to the exact requirements of each development team. For example, during a Gilb-type inspection, the inspectors search for faults during individual preparation (Gilb and Graham, 1993), while during a Fagan inspection fault-finding is left until the inspection meeting (Fagan, 1976). The tool should be capable of enforcing either process, depending on that in use by the development team.

The first stage the tool should support is planning, usually involving only the moderator. The planning stage would include entering the inspection participants and assigning their roles, as well as preparing the documents for inspection. Such preparation may involve running defect detection tools on the document, or else successful runs with these tools may be entry criteria for the inspection. If the defect detection tools are run during the planning stage of inspection, then any items found should be cast as comments for inspectors to review during the preparation stage. When the inspection has been set up, the moderator can then send invitations to the inspection participants, usually by electronic mail.

On commencing the overview, the documents used during the inspection are usually distributed, but in an on-line inspection, this is no longer necessary. The overview is also used as an introduction to the material, but as Gilb and Graham state, such a meeting (called a kickoff meeting in their terminology) is not always necessary (Gilb and Graham, 1993). However, the overview can be a convenient time to provide guidance on expected defects in the document and to set inspection targets. Although this material may be distributed electronically, it is perhaps useful to hold a meeting to ensure team morale is high. Even though our intention is to automate the inspection process, we should still keep any part of the manual process which is to our benefit.

5.3. *Individual Preparation*

A basic requirement is that the tool should allow the inspection of an on-line version of the document. This facility should be available for any type of document from source code and plain English text to dataflow diagrams and object diagrams. There should be some means of cross-referencing both within a document and across multiple documents, for example to show all instances of the use of a variable or abbreviation. There should be a means for creating annotations which are linked to the part of the document to which they refer. This may be a line or zone of text or a component of a diagram. These annotations should be capable of being given a type, indicating the purpose of that annotation or the type of fault which it describes, thereby giving more detailed metrics on the types of defects found.

Checklists or other defect detection aids used must be available on-line. Checklists should allow each item to be marked as complete and the tool should be able to report usage to provide feedback on their effectiveness. Similarly, any standards that apply to the document should be available on-line for consultation.

5.4. The Inspection Meeting

When each inspector has finished individual preparation, the inspection moves on to the group meeting, where potential defects are discussed and their disposition recorded. These defects are those found during individual preparation, and therefore a fundamental feature of the tool is for each inspector to be able to call up previously prepared comments and bring them to the attention of all participants. This is usually achieved by opening a window containing the text of the comment on each inspector's screen. The team can then proceed to discuss that comment.

Computer mediated interaction is an opportunity to assist in maintaining the meeting structure. Although the inspection meeting has a well-defined structure, it is easy for the meeting to stray from the agenda, especially with an ineffective moderator. The computer may be able to help the moderator by providing cues on how well the meeting is going. For example, if no defects have been recorded for several minutes, it may be that the inspection team are spending too much time on discussion. The computer can hint to the moderator that the meeting should move on. Similarly, lack of contribution from an inspector may be brought to the attention of the moderator, who may then try to encourage that participant.

The output from the meeting consists of a list of defects which the inspection team regard as existing in the document. This list is compiled by the recorder. The tool can assist the recorder in several ways. When an inspector proposes a comment at the meeting, the recorder should be given special controls to allow the comment to be classified for type, class and severity.

The group meeting is considered to be an important part of the inspection process as the work done by the group can be greater than the sum of the contributions from individuals. One direct result is the finding of new defects at the meeting, but there are other benefits, including the education of new inspectors. At the same time, the group meeting process can also have adverse affects on the outcome of the meeting, resulting in a loss of productivity, which can be directly measured as a lower number of reported defects, as reported by Votta (1993). One aim of automating the inspection process is to improve or maintain the gains while eliminating or reducing the losses. The sources of meeting gains and losses include those described by Nunamaker et al. (1991), but we describe them with specific reference to an inspection meeting. Additionally, because an inspection meeting is well-structured with a well-defined agenda, many of their points concerning task and process structure are of little relevance and these are omitted.

5.4.1. Gains from Inspection Meetings

There are at least five benefits that may be gained with a group meeting. The most important benefit where software inspection is concerned is that a group can provide a more objective evaluation of an idea (Nunamaker et al., 1991). This manifests itself in an increased ability to detect flaws. Incorrect suggestions tend to be rejected by a participant other than the one who made the original suggestion, as intimated by Shaw (1971). Shaw also provides evidence that overall group judgement is superior to that of the average participant. Also, there is the much quoted synergy (Nunamaker et al., 1991), which come from several people having access to each others information. This information sharing may allow one person to generate an idea which the owner of the information could not. In a similar vein, working in a group can produce more stimulation for each individual, as the desire to be seen to do well can motivate an individual. This can be seen during an inspection as a desire to find more defects than the other team members. Another gain comes from the fact that the group as a whole has more information than any single member, which can lead to producing more and better solutions to any given problem (Shaw, 1971). Group work also provides an opportunity to coach more inexperienced individuals. This is essential for such highly skilled roles as the moderator. By including trainees in the meeting, they can imitate and reproduce the skills of the more experienced participants.

5.4.2. Losses from Inspection Meetings

The first meeting loss comes from free riding, where a participant may rely on others to perform the task (Nunamaker et al., 1991). In a software inspection, this manifests itself through an insufficiently prepared inspector attending the meeting and being unable to contribute. When individual preparation is performed with an inspection tool, the tool can be used to track indicators of inspector effort such as the amount of document coverage achieved, amount of time spent in preparation and number of defects found. The moderator can use this information to exclude such individuals from the meeting, encourage them to invest more effort, or in extreme cases postpone the meeting until the inspector has prepared sufficiently.

Production blocking is another potential factor (Nunamaker et al., 1991). It is composed of three related problems. Attenuation blocking occurs when a participant has a comment to contribute but cannot do so and forgets or suppresses it. Attention blocking is the inability to think of new comments by having to concentrate on listening to others. Concentration blocking occurs when participants concentrate on remembering commments instead of thinking of new comments. These blocking effects are less relevant to the highly structured inspection meeting than they are to a more general unstructured meeting, because if the inspectors have prepared well, the meeting will simply consist of voting on and logging each issue with the minimum of discussion. However, an inspection tool can provide several important features which can help reduce these effects. Attenuation blocking can typically be reduced by the document annotation facility, which allows the inspector to make a permanent record of the comment and bring it to everyone's attention. It is also reduced by having parallel channels of communication. Attention blocking is less of an issue if most of the defects discussed at the meeting are found beforehand, such as happens in a Gilb-style inspection (Gilb and Graham, 1993), because if more than one inspector finds the defect, there is an opportunity for the non-proposing inspectors to reflect further on the document. Concentration blocking is reduced if the discussion is held electronically, as there is a permanent record of the discussion which can be viewed later. This electronic record can also reduce another possible loss where participants fail to remember the discussion.

Conformance pressure reduces a participants ability to be critical of another's comments. A tool supported inspection can easily overcome this by providing an anonymous voting mechanism. Each inspector can then feel able to vote against a comment without fear of reprisal or any other ill feeling. Air time fragmentation occurs in a traditional meeting as discussion must proceed serially, with one participant speaking at a time. An inspection support tool can reduce this fragmentation by allowing parallel communication, not only by speech but also by message sending and gesticulation, such as pointing with a cursor or highlighting text.

5.4.3. Discussion Support

Discussion can be facilitated in several ways. The simplest method is to limit the inspection to a single room to allow normal discussion. The tool would then allow each inspector to propose comments, which are then discussed and the outcome recorded by the scribe. This is the approach used in ICICLE (Brothers et al., 1990; Sembugamoorthy and Brothers, 1990). Again, there should be support for recording the comments since this is one of the most time-consuming tasks that has to be performed during the inspection meeting, as well as being error-prone. For this type of meeting, the environment is also important. Each inspector should have easy access to a workstation, yet the machines should not dominate the meeting, otherwise the effectiveness of a face-to-face meeting may be lost. The workstations should be arranged around a table in a traditional meeting layout. The meeting room may also have other traditional meeting aids, such as a whiteboard or projector.

If it is impractical to limit the inspection to a same-time, same-place format, then distributed meetings may be held, that is with inspectors sited in disparate locations. In addition to the tool being able to support such distribution, usually achieved by logging in to remote machines, there must be some means of communication between the inspectors. The least sophisticated method, in terms of integration with the inspection support tool, is videoconferencing, where audio and video discussion channels are available while the meeting is conducted. Videoconferencing can conceivably be added to any existing inspection tool with little or no work. This method can, of course, be used with a non-computer supported (paper-based) inspection.

The next level of support which is integrated with the inspection tool is some form of discussion client, like that found in Scrutiny (Gintell et al., 1993). This type of discussion mechanism simply allows participants to post text comments to the meeting in general. Any participant can then respond with reference to the original comment. The posting of comments is near instantaneous, allowing almost real time discussions to be held. A discussion client may be used in parallel with videoconferencing software to provide several paths of communication, reducing attenuation blocking.

The most sophisticated form of support involves integrating the workspace (i.e., the document under inspection and the supporting material) with some form of interaction support. One example of this is described by Ishii et al. (1994), who demonstrate several prototype collaboration systems, which combine a shared workspace with visual interaction between the participants. One particular prototype provides distributed use of graphics software for discussing diagrams. Each participant is seated at a large clear board. Onto this board are projected the image from a local computer and the image from the other participant's computer, which combine into a shared workspace. This workspace is overlaid on to a facial image of the other participant. Each participant also has an electronic pen which can be used to annotate this workspace. This arrangement provides a highly usable interaction system, aided by the ability to see the other participant. Even though the system is limited to only two participants, there are important features which could be applied to providing similar support for an inspection meeting. One obvious benefit of this system is the ability to support annotation of diagrams, and it follows that text can also be used in the shared workspace. This addresses one of the fundamental requirements we stated at the beginning. Another feature which is useful in an inspection meeting is the use of electronic pens to allow document features to be highlighted, allowing attention to be directed to specific areas of the document. It may be argued that such a pointing device is more usable than a mouse. Such pens could also be used to annotate a page with comments by storing these highlight marks with the document. This type of computer supported meeting environment probably represents the ideal solution, however the hardware required is not common. Currently, high-end PCs and workstations are widely available and in the interim it may be more profitable to make the best of these machines rather then designing for a technology which may not have such a high acceptance. We look forward to an era when every company or large institution will have a computer supported meeting room.

5.4.4. Asynchronous Inspection Meetings

An alternative to holding an inspection meeting is to employ an asynchronous inspection method. Such a method does not rely on being able to have all participants available at one time for a discussion. Instead, the inspection proceeds by inspectors creating publicly accessible comments on the document under inspection. These comments can themselves be commented on by other inspectors, allowing a discussion to take place much as happens in an electronic newsgroup. In addition, it is possible to provide a mechanism for voting on the status of a comment. An example of such a review method is FTArm (Johnson and Tjahjono, 1993). An asynchronous inspection is a credible alternative to a synchronous meeting as it can remove many of the meeting losses described. For example, asynchronous discussion increases parallelism in the process, as multiple threads of discussion can continue at the same time. This also reduces production blocking as inspectors are allowed to concurrently present their comments. The discussion system also provides a meeting memory, allowing inspectors to review the discussion so far and make an appropriate comment. In addition to reducing potential loss factors, asynchronous inspection preserves the gains achieved through involving several people in the defect-finding process. The group as a whole still has more information than any single participant and the group is more capable of spotting defects in ideas. There is also a good learning environment in watching these discussions progress, with the trainee able to follow such discussions at their own pace. Group synergy and stimulation is also present. However, an asynchronous meeting is usually followed up with a face-to-face meeting to resolve any remaining topics (Johnson and Tjahjono, 1993). This meeting will still suffer from the problems described before. Nevertheless, since asynchronous meetings are a feasible alternative to a traditional meeting, we feel that an inspection support tool should provide both modes of meeting.

5.5. *Follow-up Stage*

Whichever meeting format is chosen, when the meeting is complete the producer must make changes to the document to address any defects found. There are two possible ways in which computer support can be used here. First of all, given an electronic defect list, the producer should be able to mark a response to each defect, indicating if the document was changed to take account of the defect, or if unchanged then the reason why should be stated. Secondly, the change should be implemented using a change control tool integrated with the inspection support tool. This allows changes to be related to the inspection which suggested them, allowing improved defect tracking and better evaluation of both the inspection process and the software development process.

5.6. *Metrics Collection*

Data from the inspection process is an important feedback to help improve the software development process. For this reason, a vital area of computer assistance is in producing metrics from the inspection. The data from which these metrics are generated is traditionally collected by hand. Not only is this time-consuming, with the duty usually falling to the recorder, it is also very error-prone. Automated collection of this data removes both these problems and also provides more finely-grained data. For example, an inspection tool can record the amount of time spent in preparation by each inspector, as well the amount of the document covered. When this data is coupled with the number of defects found, it is possible to set guidelines on the optimal amount of time which should be used for inspecting a document of a similar type.

Further use can be made of metrics to help improve upstream processes. A large number of occurrences of a certain type of defect may indicate a weak point in the development process. By making the developers aware of the defect they can take steps to improve the process at the point where the defect is injected into the document.

6. Conclusions

Inspection is believed to be the most cost-effective technique for finding defects in documents produced during the software lifecycle. However, it has been found to be difficult to implement and is labour intensive. It is therefore a prime target for computer support. This paper has described current approaches to computer support to assist software inspection. While these approaches tackle some of the main issues in supporting inspection, there are areas which have been neglected. One example is the lack of support for diagrams, others include sparse distributed meeting support and lack of integration with existing CASE tools.

With such deficiencies in mind, we have presented an informal specification of a well-featured inspection support tool. First of all, the system should model all the roles of the inspection participants, with the features of the tool available to each participant dependent on their role. During the planning stage, the tool can help the moderator select the documents to be inspected, invite participants and run any required defect finding tools. For individual preparation, there should be support for browsing and annotation of any type of document and cross-referencing both within and between documents. The system is also required to support the use of checklists, as well as provide any relevant standards. For the inspection meeting, we require the system to support three meeting types. In both same-time, same-place and same-time, different-place meetings, a minimum requirement for the tool is to provide facilities for proposing and voting on defects. If the meeting is held in a distributed fashion then it is also necessary to provide some form of discussion support, either using a messaging system or through videoconferencing. There is also much scope to make use of more advanced distributed co-operative work systems. The third type of meeting which can be supported is an asynchronous meeting (either same-place or different-place). This can be facilitated by allowing threads of discussion to be maintained through a comment recording system. When the meeting is complete, the tool should provide support to the author when making the required changes to the document by recording which defects have been addressed and which have not. There is also scope for integration with change control tools for this phase. Finally, metric gathering throughout the inspection is another essential requirement. Metrics gathered will typically include quantities of defects found and time spent executing the inspection. These metrics can then be used to fine-tune the inspection process.

This list of requirements is derived from features found to be useful in currently available inspection tools, along with facilities required in an electronic meeting support system. It also takes into account some weaknesses in currently available tools and our own experience with several of these tools. However, it is important to realise that such features must be evaluated for suitability before their inclusion in an inspection support tool is deemed essential. This evaluation may be quite complex as there are many interdependencies between features that must be accounted for, precluding the evaluation of features in isolation. Even given these difficulties, we feel that all these areas must be addressed for a computer-assisted inspection to provide substantial gains over a manual inspection. Such gains are the rationale behind our desire to automate the inspection process, yet due to the lack of some key features, the current generation of inspection support tools may not be capable of providing these gains. As far as we are aware, there is no definitive study on the effectiveness of manual inspection in comparison with tool-supported inspection. To this end, our future research lies in implementing a prototype to evaluate the features described.

Acknowledgments

We are extremely grateful to those who provided us with versions of their inspection support tools, in particular Gérard Memmi of Bull HN Information Systems, Inc. U.S. Applied Research Laboratory, who supplied us with Scrutiny, and Eric Jung of Bell Communications Research, who supplied us with ICICLE. We also thank the authors of CSRS, who have made

a version publicly available, details of which can be found at `http://www.ics.hawaii.edu/~csdl/egret/release-notes.html.`

References

Ackerman, A. F., Buchwald, L. S., and Lewski, F. H. 1989. "Software Inspections: An Effective Verification Process," *IEEE Software*, (6)3:31-36.

Brothers, L., Sembugamoorthy, V. and Muller, M. 1990. "ICICLE: Groupware for Code Inspections," In *Proceedings of the 1990 ACM Conference on Computer Supported Cooperative Work*, ACM, October, pp. 169-181.

Doolan, E. P. 1992. "Experience with Fagan's Inspection Method," *Software - Practice and Experience*, (22)2:173-182.

Fagan, M. E. 1976. "Design and Code Inspections to Reduce Errors in Program Development," *IBM System Journal*, (15)3:182-211.

Fagan, M. E. 1986. "Advances in Software Inspection," *IEEE Transactions on Software Engineering*, (12)7:744-751.

Gilb, T. and Graham, D. 1993 *Software Inspection*. Addison-Wesley.

Gintell, J. W., Arnold, J., Houde, M., Kruszelnicki, J., McKenney, R. and Memmi, G. 1993. "Scrutiny: A Collaborative Inspection and Review System," In *Proceedings of the 4th European Software Engineering Conference*.

Humphrey, W. S. 1989. *Managing the Software Process*, Addison-Wesley,

Ishii, H., Kobayashi, M. and Arita, K. 1994. "Iterative Design of Seamless Collaboration Media," *Communications of the ACM*, (37)8:83-97.

Johnson, P. and Tjahjono, D. 1993. "CSRS Users Guide," Technical Report ICS-TR-93-16, Department of Information and Computer Sciences, University of Hawaii.

Johnson, P. 1994a. "An Instrumented Approach to Improving Software Quality Through Formal Technical Review," In *Proceedings of the 16th International Conference on Software Engineering*, May.

Johnson, P. 1994b. "Supporting Technology Transfer of Formal Technical Review Through a Computer Supported Collaborative Review System," In *Proceedings of the 16th International Conference on Software Quality*.

Knight, J. C. and Meyers, E. A. 1991. "Phased Inspections and their Implementation," *Software Engineering Notes*, (16)3:29-35.

Knight, J. C. and Meyers, E. A. 1993. "An Improved Inspection Technique," *Communications of the ACM*, (11)11:51-61.

Macdonald, F. and Miller, J. 1995 "Modelling Software Inspection Methods for the Application of Tool Support," Technical Report RR-95-195 [EFoCS-16-95], Department of Computer Science, University of Strathclyde.

Mashayekhi, V., Drake, J. M., Tsai, W. T. and Reidl, J. 1993. "Distributed, Collaborative Software Inspection," *IEEE Software*, (10)5:66-75.

Mashayekhi, V., Feulner, C. and Reidl, J. 1994. "CAIS: Collaborative Asynchronous Inspection of Software," In *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 21-34.

Nunamaker, J. F., Dennis, A. R., Valaich, J. S., Vogel, D. R. and George, J. F. 1991 "Electronic Meeting Systems to Support Group Work," *Communications of the ACM*, (33)2:40-61.

Reidl, J., Mashayekhi, V., Schnepf, J., Claypool, M., Frankowski, D. 1993. "SuiteSound - A System for Distributed Collaborative Multimedia," *IEEE Transactions on Knowledge and Data Engineering*, (5)4:600-610.

Russell, G. W. 1991. "Experience with Inspections in Ultralarge-Scale Developments," *IEEE Software*, (8)1:25-31.

Sembugamoorthy, V. and Brothers, L. 1990. "ICICLE: Intelligent Code Inspection in a C Language Environment," In *Proceedings of the 14th Annual Computer Software and Applications Conference*, pp. 146-154.

Shaw, M. E. 1971. *Group Dynamics: The Psychology of Small Group Behaviour*, McGraw-Hill.

Votta, L. G. 1993. "Does Every Inspection Need a Meeting?" In *Proceedings of the 1st ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 107-114.

Weller, E. F. 1993. "Lessons from Three Years of Inspection Data," *IEEE Software*, (10)5:38-45.

Automated Software Engineering 3, 219–237 (1996)
© 1996 Kluwer Academic Publishers. Manufactured in The Netherlands.

Design by Framework Completion

DIPAYAN GANGOPADHYAY dipayan@watson.ibm.com
IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598

SUBRATA MITRA mitra@vnet.ibm.com
Application Development Technology Institute, IBM Software Solutions Division, 555 Bailey Avenue, San Jose, CA 95141

Abstract. An *object-oriented framework* in essence defines an *architecture* for a family of applications or subsystems in a given domain. Every application in the family obeys these architectural restrictions. Such frameworks are typically delivered as collections of inter-dependent abstract classes, together with their concrete subclasses. The abstract classes and their interdependencies implicitly realize the architecture. Developing a new application reusing classes of a framework requires a thorough understanding of the framework architecture.

We introduce an approach called "Design by Framework Completion", in which an *exemplar* (an executable visual model for a minimal instantiation of the architecture) is used for documenting frameworks. We propose exploration of exemplars as a means for learning the architecture, following which new applications can be built by replacing selected pieces of the exemplar. For the piece to be replaced, the inheritance lattice around its class provides the space of alternatives, one of these classes may be suitably adapted (say, by sub-classing) to create the new replacement.

"Design by Framework Completion" proposes a paradigm shift when designing in presence of reusable components: It enables a much simpler "top-down" approach for creating applications, as opposed to the prevalent "search for components and assemble them bottom-up" strategy. We believe that this paradigm shift is essential because components can only be fitted together if they all obey the same architectural rules that govern the framework.

Keywords: framework understanding, component-based software engineering, software reuse, architecture, representation, executable model, learning by example

1. Introduction

This paper concerns reuse driven application development, in particular, in the presence of object oriented application frameworks[1]. A framework (Deutsch, 1989) is designed to cover a family of applications or subsystems in a given subject area (domain) and is typically delivered as a collection of inter-dependent abstract classes (together with a set of concrete classes specializing the abstract ones). The abstract classes together with their dependencies (either structural, e.g., a container-contained relationship; or behavioral, such as patterns of communication) in effect realize the architecture of the framework, while concrete classes provide known variations, yet obeying the architectural rules. The task of the application designer is to specialize the abstract classes and instantiate these specialized (concrete) classes as objects to create a desired application (i.e., a new member of the application family). Since these new applications must necessarily obey the architectural rules, the biggest challenge in this endeavor is to understand the architecture, without which instances of the desired concrete classes cannot be made to work together.

Consider Graphical User Interface (GUI) as an application domain. A GUI framework will stipulate that every window will consist of a menu bar, a tool palette, and a canvas showing the contents of the window. Gaining competence with such a framework requires understanding the responsibilities of the abstract classes, the communication protocols among them, and the essential virtual operations left open for the subclasses to define. In the GUI framework, for example, a "widget" is responsible for the shape that gets rendered, while the canvas may determine the placement of the widget. Second, a canvas has a container-contained (structural) relationship to its contained widgets. Third, when a window is exposed, the canvas communicates this to the individual widgets, each of which then redraws its shape—this exemplifies the behavioral relationship (i.e., the message passing protocol) between the canvas and its widgets. Finally, the (abstract) widget class may leave rendering of the actual shape up to its concrete subclasses (e.g., rectangle, etc.). It is impossible and even meaningless to reuse the widget without understanding all of these relationships. In other words, such relationships between the abstract classes, i.e., window, canvas, tool palette, menu bar and widget, define the essential architecture of all GUI applications permissible using such a framework.

Most existing approaches for software reuse assume a bottom-up component assembly paradigm see (Goldberg and Robson, 1984; Pietro-Diaz and Freeman, 1987; Gangopadhyay and Helm, 1989; Maarek, 1990; Rollins and Wing, 1991). We believe that such approaches would be successful only if the components under consideration are more or less self-contained (such as large software modules), and where one does not have to modify them for reuse. Some of these existing approaches have also been used in situations of framework-based reuse, where, they focus on finding and selecting a class at a time out of a framework library, using various techniques for browsing and search, such as, specification based retrieval schemes, information retrieval based on documentation, or some sort of faceted classification. All of these techniques focus on one class at a time rather than on the inherent architecture behind the ensemble of classes. In short, they treat frameworks just like any other class library. Thus, the unaided reuser ends up spending an enormous amount of time trying to select and assemble architecturally compatible sets of components, either by trial and error or by chasing source-code of the method definitions and reading informal class-by-class documentation. Fitting components together in a bottom-up fashion is thus as hard as solving a Jigsaw puzzle! However, since puzzle pieces are made to fit an outline, why not start with the outline itself? The above intuition is depicted in figure 1.

In this paper, we argue that a paradigm shift is needed—instead of constructing applications bottom-up by assembling fine-grained components, one ought to apply the "Design by Framework Completion" approach, in which a new application is constructed top-down by understanding and adapting an example application. For example, if one is interested in constructing a graphical editor, it makes much more sense to adapt an existing one, rather than to create one from scratch by assembling pushbuttons, scrollbars, menus, and so forth. In other words, we are proposing creation of new instances of the jigsaw puzzle by

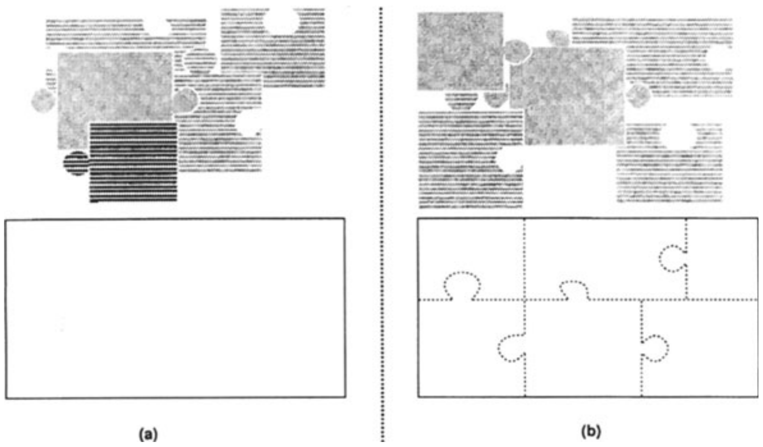


Figure 1. (a) Bottom-up assembly: Putting pieces together without knowing the architecture is like solving the jigsaw puzzle from scratch. (b) Given the outline pattern, the puzzle is easy to solve, as we know what piece could fill which slot of the pattern.

replacing pieces from a given and completed puzzle, as shown in figure 2. For frameworks, the architecture (an abstract entity) is like the jigsaw outline, while the example application is the given (complete) puzzle, which instantiates the abstract architecture.

In the "Design by Framework Completion" approach we use an exemplar as a executable visual model for a minimal instantiation of the architecture of a framework, and propose exploration of exemplars as a means for architecture understanding. Following such exploration, new applications can be built by replacing selected pieces of the exemplar. For each selected piece, the inheritance lattice around its class provides the space of alternatives. The reuser may either use one of these classes directly, or could suitably adapt (say, by sub-classing) one of them to create a new replacement. The steps may be repeated until the reuser is satisfied with the modified application.

An exemplar is an executable visual model consisting of instances of concrete classes together with explicit representation of their collaborations. For each abstract class in the framework, at least one of its concrete subclasses must be instantiated in the exemplar. Therefore, the number of elements in an exemplar is of the order of the number of abstract classes in the framework. Since even large frameworks have only a few abstract classes, a small number of instances suffice in creating an exemplar.

Based on a visual executable modeling technology, called ObjChart (Gangopadhyay and Mitra, 1993; Gangopadhyay et al., 1993), we have created an environment (Gangopadhyay, 1994) to carry out model level exploration of exemplars. (The main constructs of this modeling notation appear in figure 10, in the Appendix.) Model level exploration is unique among the prevalent approaches to framework based development, which still emphasize different class browsing and retrieval technologies, active cookbooks, or code tracing. Some problems of bottom-up assembly of components have been mentioned independently by Garlan et al. (1995). Detailed comparison is deferred until Section 3.

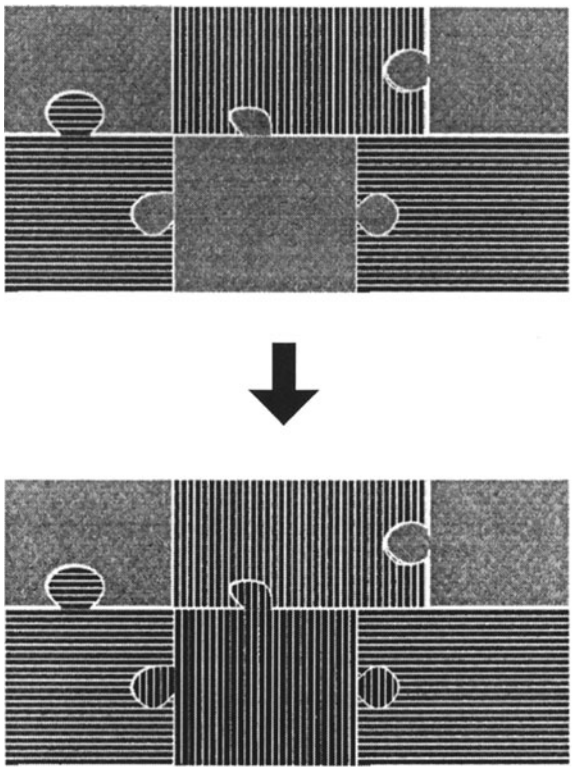


Figure 2. Design by framework completion.

2. Design by framework completion approach

In this section, the approach of Design by Framework Completion is illustrated via an example of reusing a "Persistence Framework".

Assume that a reuser wants to design an application for persistently storing multi-media compound documents on some storage device, in particular storing audio clippings. To accomplish this, a Persistence Framework[2] is chosen. The Persistence Framework is designed to store and retrieve any in-memory compound object into persistent storage devices. Over and above the usual mapping of in-memory structured data into byte-streams in storage devices, the framework allows customization of clustering of related objects and data-compression facilities.

We will go through a scenario where, in order to store lengthy audio-clippings, the reuser will have to use specialized data-compression algorithms. The essential intellectual task for the reuser is to pin-point the class in the framework where the specialization should take place.

For the rest of this section, we will first briefly describe the example problem and then take the reader through the steps of Design by Framework Completion approach via screen-shots. We deliberately keep the initial description of the framework brief, with the hope that the reader will intuitively understand the details of the framework, just as a reuser would do in the ObjChart Environment.

2.1. Problem description

The framework consists of a storage manager storing persistent copies of "complex objects". Complex objects have nested structure, i.e., each object may be contained inside another object. Essentially, compound documents are examples of such objects.

There are two design points in this framework:

1. In order to optimize storage and retrieval it provides clustering of related objects into groups (called IOGroups). Whenever any persistent object in the group is written into I/O media, so are the other objects in the group. However, because the optimal clustering for retrieval efficiency depends upon the navigation pattern of specific application programs, the specific clustering policy is left open for customization.
2. Structured data contained by any given in-memory object has to be linearized into byte-streams before writing to I/O media. However, because the specific stored format depends upon the object type (can range from just data structures to video-clippings requiring compression), this linearization policy is also left open for customization.

For this example, we would assume that a reuser needs to create persistence services for multi-media documents, which could contain audio clippings. In this case, the reuser may need to have a compression algorithm to reduce the storage requirement. The task could be accomplished by subclassing the **formatter** class to one which can do specialized data compression. The problem for the reuser is: how does the reuser know where to make the necessary changes?

Furthermore, to store the audio clippings, e.g., on a Digital Audio Tape device, it will be necessary to have a special media object used by the IOGroup. Therefore, the reuser has to subclass the IOGroup in concert. How does he figure out this dependency?

In the rest of the section, we describe how the reuser understands the architectural relationships among the components of the framework.

2.2. (Step 1) Understanding the exemplar

The reuser starts by understanding an exemplar supplied with the framework. In this case, the exemplar shows an working example of using the framework for storing an in-memory compound object (not multi-media) into a storage device. For understanding of the exemplar, the reuser explores both the structure of the exemplar (static description) as well as the dynamics. The visual model of the exemplar is described in ObjChart notations (see discussion in Appendix for details on the ObjChart notation).

2.2.1. Exploring the structure. The overall static structure of the exemplar is shown in figure 3, using the Object Model Diagram facility of ObjChart. From this figure, the

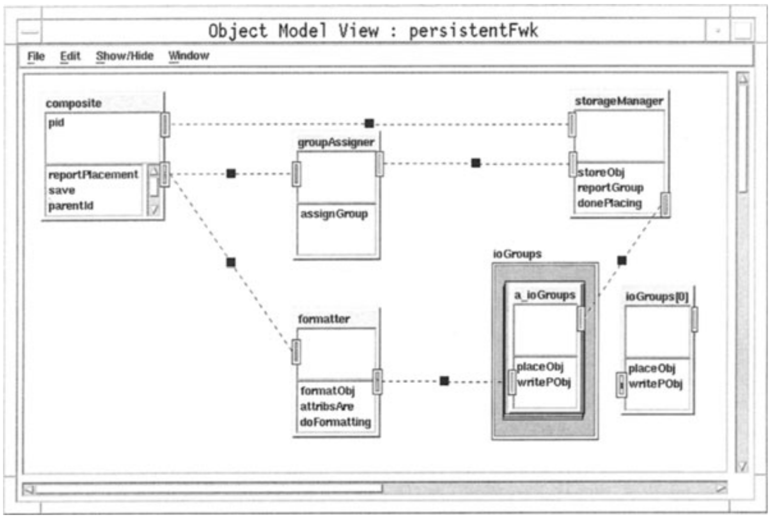
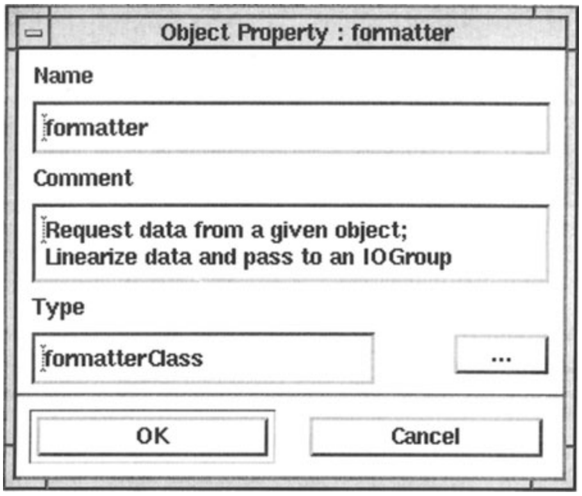


Figure 3. Exemplar for the persistence framework.

reuser sees first an overview of the constituent objects and their structural relationships. By examining the comment text associated with each object, the reuser can understand the responsibilities of each of them. For example, the comment text field for **formatter** is shown in figure 4. The objects and their responsibilities are outlined below:

1. **Composite** is a complex object, containing other objects.
2. **StorageManager** stores and restores objects.
3. The sequence **IoGroups** is a collection of IOGroups managed by the storage manager. Each IOGroup clusters a collection of persistent objects, called **persistentObjects**[3]. In particular, the object **ioGroups[0]** is an IOGroup in which **composite** is placed.
4. **GroupAssigner** determines the specific IOGroup to store a given object. This particular assigner uses the policy of placing an object in the same IOGroup as the object's container.
5. **Formatter** looks up from a given object the data to be stored, linearizes it according to its own algorithm and passes the linearized data to be written to a specific IOGroup.

In short, the Object Model Diagram of the exemplar depicts the structural relationships among the objects. Structural relationships include association, which is shown in figure 3, and containment, which can be seen by exploring the exemplar. Over and above these structural relationships, the dynamic behavior of each object is modeled using a collection of Event-Conditions-Actions (causal) rules or a Finite State Machine (FSM). (See the Appendix for more details on the behavior description mechanism of ObjChart.)


Object Property : formatter

Name

formatter

Comment

Request data from a given object;
Linearize data and pass to an IOGroup

Type

formatterClass

Figure 4. Property view of the **formatter**.

2.2.2. Exploring dynamic behavior. Having understood the roles and responsibility of each object from the structural representation of an exemplar as discussed above, the reuser now engages in understanding the dynamic collaborative relationships among these objects. This is accomplished by interactive exploration of the executable model of the exemplar. Such exploration includes observing the messaging interactions among objects on different stimuli, thereby gaining perceptual understanding the collaborative relationships (Helm et al., 1990).

For the Persistence Framework, having browsed the static structure of the exemplar model, the reuser determines a specific set of objects whose messaging interactions are of interest. At this stage, the model can be executed interactively for its usage scenarios. For this purpose, a pre-defined *tryMe* message is provided at the top level (**persistentFwk**).

When the user issues the *tryMe* message, ObjChart generates the resulting messages as an Event Scenario Trace, as shown in figure 5[4]. From the trace, the reuser can see the following messaging interactions:

- When the **storageManager** gets the message *storeObj* to store a specific object, it asks the attached **groupAssigner** to get an IOGroup. When the storage manager receives IOGroup information (through the *reportGroup* message), it requests the corresponding group (in this case **ioGroups**[**0**]) to store the object.
- A group, when asked to place an object into persistent store, sends the request *formatObj* to the attached formatter to linearize the data of the object.
- The **formatter** asks the **object** to report its attributes to be made persistent, using the *reportAttribs* message. When an object reports its attributes (using the *attribsAre* message), **formatter** linearizes the attribute values into a data stream which is passed to the IOGroup (**ioGroup**[**0**]) to write into a persistent object.
- When a group receives formatted data (through the message *writePObj*), it sets up a new **persistentObject** (in this case, **persistentObjects**[**1**]) and asks this persistent

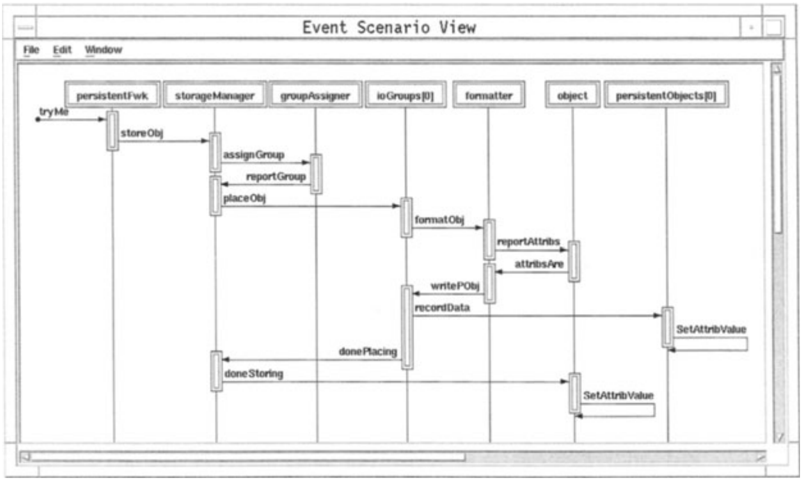


Figure 5. Event scenario trace for the persistence framework.

object to record the data; it also sends the *donePlacing* message to the storage manager.

- The **persistentObject**[**1**], on receipt of *recordData*, stores the linearized stream.
- Once its data has been written to persistent store, the storage manager sends an acknowledgement (*doneStoring*) to the object. At this point the object stores the "Id" of the IOGroup in which its data got saved.

2.3. *(Step 2) Selecting the object to be replaced*

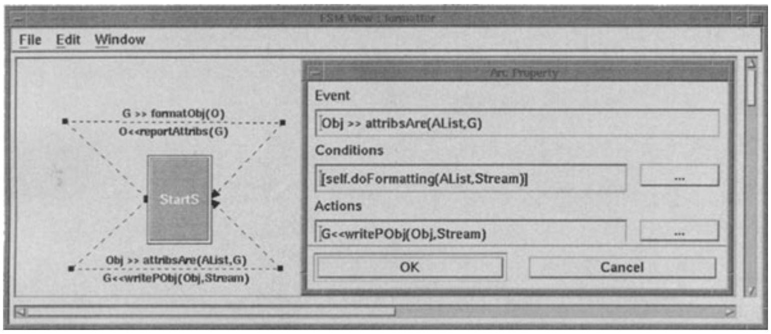
Once familiar with the overall structure and behavior of the exemplar, the reuser decides on the objects of the exemplar that need to be changed to fit the requirements.

For the Persistence Framework, the reuser would want to examine the formatter object in more detail. On examining the causal rule for *attribsAre* (as shown in figure 6) in the FSM of the formatter it becomes evident that the bulk of the formatting is done by the *doFormatting* operation.

2.4. *(Step 3) Finding alternatives from the class lattice*

For each object selected in the previous step, the reuser goes to its class and explores the class lattice (inheritance hierarchy) around this class. The class lattice provides the space of alternatives to choose from. If there is already a class to fit the purpose, the job is done. Otherwise, the reuser has to embark on creating a new class.

For the example at hand, it would be the next logical step for the reuser to try and replace the simple formatter with one which linearizes and compresses the data. To find a suitable



Event

Obj >> attribsAre(AList,G)

Conditions

[self.doFormatting(AList,Stream)]

Actions

G<<writePObj(Obj,Stream)

Figure 6. Finite state machine of the **formatter** with a causal rule.

formatter, the reuser first looks up the class name of the **formatter** object from its property window (figure 4) and locates the class (**formatterClass**) in the Class Model Diagram[5] (figure 7) for the framework. In order to find a formatter class with data compression capability, the reuser can now browse the classes in the inheritance lattice rooted at the abstract superclass (**formatterAbstractClass**) of the **formatterClass**. In this example, indeed the **formatterNCompressionClass** will do the job.

Notice that, in our approach, since the user selects the desired object that has to be replaced, and since the object already has class information, finding alternative classes is simple; much more so than having to browse through a huge collection of classes.

2.5. *(Step 4) Adaptation*

If a class with the desired properties does not exist in the class-lattice, a new class can be created by subclassing an existing one and providing new methods, etc. The desired class is then instantiated as an object to replace the corresponding one selected in the step entitled "Selecting the Object to be Replaced".

For our example, the concrete class **formatterNCompressClass** is instantiated in the object model diagram, i.e., the exemplar, and the new instance replaces the old **formatter**. At this point, the reuser has the new application, which can be immediately executed to his satisfaction. Thus, the reuser has accomplished "Design by Framework Completion"!

In order to write persistent objects into Digital Audio Tape device, the reuser can subclass a IOMedia class (not described in this paper) of the framework. The approach to identify the IOMedia class as the one to subclass from, can be achieved by the same steps described in the preceding sections.

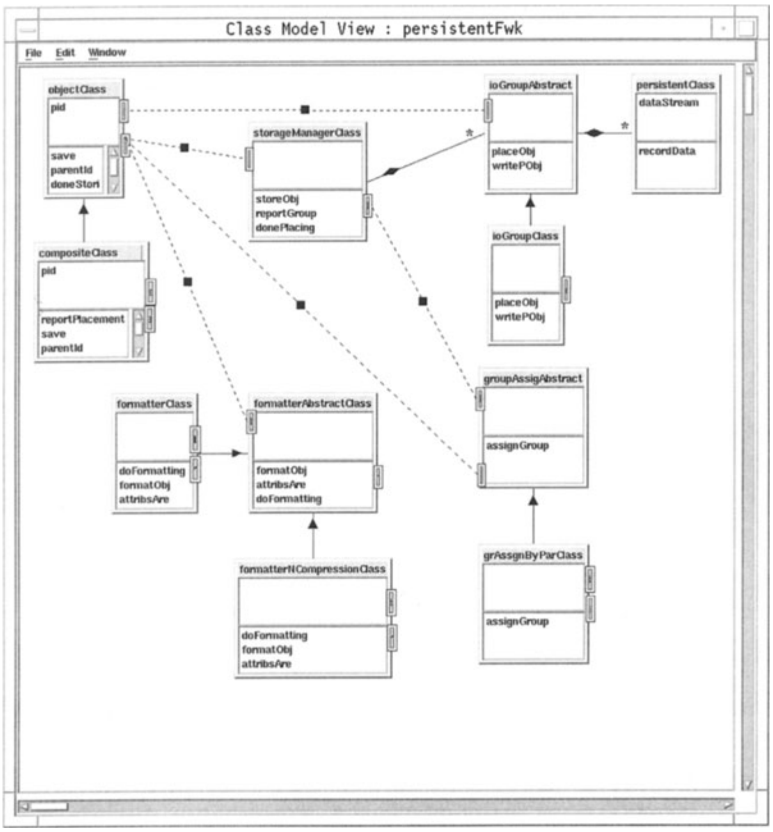


Figure 7. Class diagram for the persistence framework.

2.6. *Summary*

The preceding example illustrated our approach. Instead of trying to compose components bottom-up, "Design by Framework Completion" encourages a top-down scheme. The exemplar model shows statically the essential objects and their dependencies, and therefore provides a "footprint" of the architecture of the framework. Furthermore, the reuser not only understands the static interconnections between objects of the exemplar, but also their dynamics of messaging via execution of the model. The net result is that the reuser did not have to search through a sea of classes to find the solution; exploration of the exemplar naturally got the reuser focused on the formatter object. Thereafter, finding the suitable class was a matter of manual browsing of a small section of the class lattice.

Based on the example of the Persistence Framework, our top-down solution to reuse in presence of a framework starts with a representation scheme: Framework developers provide exemplars, which are executable visual models that minimally instantiate the architecture of the framework. An exemplar consists of one or more instances of at least one concrete class for each abstract class in the framework. The visual model will represent the structural and behavioral relationships among these objects.

Once we have the representation, we use the following recipe to accomplish "Design by Framework Completion":

1. **Exploration**. A reuser will first interactively explore the exemplar in order to understand the responsibilities and relationships of the objects. Interactive exploration include understanding of structural relationships and behavioral interactions.
2. **Selection**. Having understood how the exemplar works, the reuser decides on the objects of the exemplar that need to be changed to fit his requirements.
3. **Finding alternatives**. The reuser has to find an alternative from the class-lattice around the class of each object to be replaced. Either a ready-made solution is already available, or a new alternative has to be created by adapting an existing class.
4. **Adaptation**. A new class can be created by subclassing an existing one and providing new methods, etc. The new class is then instantiated as an object to replace the corresponding one selected in step 2.
5. The steps 1–4 are repeated until the reuser is satisfied.

A few attempts has been made in aiding framework based development, using cookbooks, for specific frameworks; for example (Schappert, 1994). Cookbooks aim to guide reusers through specific tasks in a prescriptive manner. Using "wizards" (Soetarman, 1994) is another idea where a reuser is guided to fill in a template suited to a specific kind of application. Our use of an exemplar is akin to adapting a template. However, we believe that our approach of understanding through active exploration gives the reuser a fundamental understanding of the relevant dependencies, which is not achievable through pre-defined or prescriptive steps.

3. Discussion

In this paper we have advocated the need for understanding the architecture of a framework by interactive exploration as a necessary step towards framework based reuse, as opposed to the prevalent notion of component by component, bottom-up integration. The essential technical ingredients of our approach are

- Learning by example,
- Explicit representation of the architecture, and
- Interactive exploration of the architecture.

In the remainder of this section, we compare our approach with others known in the literature.

3.1. Code tracing vs. executable models

Our approach enables architecture understanding by interactive exploration of a minimal example, i.e., an exemplar. Learning something abstract via an example, is a well-known learning technique in the AI and cognitive psychology (Fischer, 1987) literature.

Naturally, one may argue that exploration of a code-level example program, in lieu of executing the visual model of an exemplar, might be as effective. Schemes like "Active Programming", (Rosson and Carroll, 1993) and "Program Visualization" (De Pauw et al., 1993), in fact, use code animation and tracing techniques and could be applied for understanding a code-level example program.

The major drawback with code level animation approaches is that they have to instrument "source code", which may not be readily available. Furthermore, the volume of trace that is usually generated in such methods is overwhelming and therefore helps little in clearly and succinctly understanding the behavioral protocols implicit in the abstract classes.

ObjChart models are at a higher level than code. For the Persistence Framework, for example, the exemplar has 7 ObjChart objects abstracting several thousands of lines of code. Furthermore, ObjChart objects in an exemplar can be attached to compiled code-level classes (Gangopadhyay, 1994) of the framework, that is, execution mixes interpretation of visual models with execution of dynamically loaded binaries. During exploration, such objects, on receipt of a message, execute both its FSM and methods of the attached code-level classes. Therefore, the traces are generated at the model level, even while executing real code.

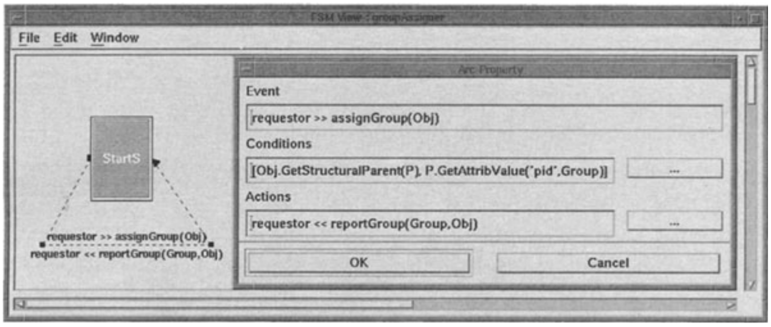
Finally, there is an emerging consensus that design-reuse is more effective than code-reuse. Our approach of dealing with a framework at its architectural level is consistent with this direction.

3.2. Accurate documentation of framework architecture

Our approach relies on interactively explorable documentation of framework architecture. And the ObjChart notations with their precise executable semantics is the vehicle of accurate documentation.

The key problem in framework architecture documentation is to accurately capture the *collaboration relationships* among the classes, not just the description of classes and their structural relationships. Unless the collaborations between the abstract classes are made explicit, the reuser encounters a daunting task of inferring such collaborations by reading source code. By using multi-party protocol objects as first-class entities (Helm et al., 1990; Gangopadhyay and Mitra, 1993), ObjChart notations capture such collaboration relationships accurately. A protocol object has a port for each of the participant in the protocol and a FSM which describe the progress of the protocol in reaction to receipt of the messages through its ports. As an example, the FSM of the **groupAssigner** object, shown in the figure 8, depicts the encapsulated protocol between the participants **storageManager** and **composite** connected to its ports. See the appendix of this paper for a brief description of the ObjChart constructs.

The community of framework developers are becoming increasingly aware of the need for representing framework architecture (IBM Publication No Z123-7461-0, 1995) and patterns implicit in the frameworks (Pree, 1994; Gamma et al., 1994). Using a combination of structural relationships (Rumbaugh et al., 1991) amongst classes, and interaction diagrams (Jacobson et al., 1992) they attempt to document the architecture. However, their representation schemes do not accurately capture the inter-component relationships. For example, in the Gamma et al.'s (1994) representation of the Abstract Factory pattern, the so-called



Event

requestor >> assignGroup(Obj)

Conditions

[Obj.GetStructuralParent(P), P.GetAttribValue('pid',Group)]

Actions

requestor << reportGroup(Group,Obj)

Figure 8. Finite state machine of the group assigner.

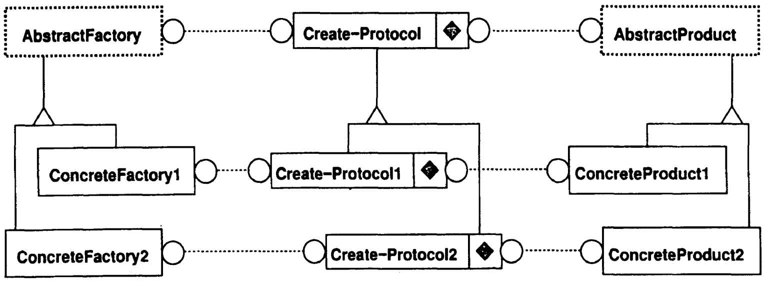


Figure 9. ObjChart model of the abstract factory pattern.

"creational" links are hardwired between a concrete factory class and the corresponding concrete product classes. Therefore, if one has to sub-class any of their concrete factories, it is unclear which concrete products this factory sub-class is allowed to create. The essential problem is the lack of support for *inheritance among relationships*. ObjChart, on the other hand, treats *relationships as first-class entities*, and hence, inheritance among them is permitted. In the ObjChart model of the Abstract Factory, as shown in figure 9, the (abstract) creation link between the abstract factory and the abstract product classes is represented as a protocol object. The corresponding protocol between a concrete factory and its concrete product classes simply inherits from the abstract one.

More importantly, given the informality of these representation schemes, they are not amenable to tool-assisted interactive exploration of dynamics from multiple perspectives. We believe that such exploration is absolutely essential for gaining insight into the architecture. The English descriptions and static interaction diagrams, as used in these representation schemes, at best permit static documentation only. In ObjChart, we encapsulate protocols themselves as reusable objects, and protocols are specified either by a set of interaction diagrams or, more precisely, by using FSMs. The ObjChart environment, being capable of interpreting these interaction diagrams and FSM descriptions, allows simulation and visualization of these protocol specifications from different perspectives.

3.3. Overhead for creating exemplars

Creating exemplars for existing frameworks introduces overheads beyond the actual code base. For example, pertinent questions may be: given the 2,000 or so classes of the Taligent frameworks, how much effort is needed to create the necessary exemplars? How large does each exemplar need to be?

First of all, experience shows that a good framework architecture has necessarily only a few abstract classes, even with numerous concrete subclasses. Our exemplars need only a single instance of any concrete subclass for each abstract class; therefore, only a few elements suffice. An exemplar is skeletal, emphasizing the abstractions. Its purpose is to provide an entry-point to the major abstractions and their many possible variations (the concrete subclasses).

Second, creating exemplars is no more of an overhead than writing user-manuals for frameworks, which the reusers need in any case. In fact, these exemplars are "live" documentation since ObjChart is an executable specification language.

Finally, creating good frameworks require a substantial amount of design effort, since frameworks have to be abstract and must also localize carefully the responsibilities in view of future variations. As such, they can very well use an OO analysis and design tool with execution capability, such as ObjChart. That way, the models of frameworks would be available naturally, without any extra effort. Thus, we are optimistic that new frameworks will be provided with their visual models.

3.4. Deemphasizing the retrieval and browsing problem

Most existing approaches for software reuse emphasize the problem of finding a suitable component fitting the need. They are extracting one component at a time. [Faceted classification scheme (Pietro-Diaz and Freeman, 1987), class hierarchy browsers (Goldberg and Robson, 1984), lexical affinity based information retrieval (Maarek, 1990), and specification of components as search keys (Gangopadhyay and Helm, 1989; Rollins and Wing, 1991) are examples.] However, there are a few major drawbacks with these approaches:

1. Reusing one component at a time implies that there is no support for understanding (possibly hidden) dependencies between components. Often times, there may be subtle dependencies (such as components which could only work in pairs) which gets in the way of bottom-up reuse efforts starting with a component at a time.
2. Since the query issued by a reuser is matched against the component description provided by the designer, vocabulary mismatch (i.e., the mismatch in the vocabulary used in the query and the component descriptions) becomes a serious hindrance to the success of the scheme. Empirical evidence shows that incorporation of synonym dictionaries does not alleviate the problem. Even with faceted classification schemes with limited vocabulary,

the problem lies still in establishing a shared understanding of the meanings of these words.

3. Finally, even when we ignore the vocabulary mismatch problem, there arises a need to ensure that the specification of the query must "match" that of some component from the class library. Such specification subsumption are computationally hard and, in general, undecidable.

Our approach solves the first problem. The reuser can get, rather easily, a complete insight into the architecture of the entire family of applications covered by the framework, by exploring an exemplar. As regarding the second problem, the selected component provides the specification which gets used in finding alternatives from the component library (they both use the designer's vocabulary), and the search is limited to only a small portion of the class lattice. Therefore, vocabulary mismatch is not as severe as it would be for an arbitrary query over the entire class library. Finally, an inheritance lattice by definition stipulates that a subclass does at least as much as its superclass. Because in our "Design by Framework Completion" approach alternatives are chosen from the inheritance lattice, the problem of computing specification subsumption does not arise.

In essence, we believe that component retrieval is not the problem in reuse, but to fit them together is the bigger issue (Gangopadhyay, 1991).

3.5. Design for enhanced reuse

We believe that the key to good framework design is explicit and localized representation of the protocols between the components of the framework, rather than distributing such protocols inside the method bodies of these components. Explicit representation enables understanding while localization allows ease of refinement of the protocols themselves.

Our example of Section 1, the model of the Persistence Framework (figure 3), uses two such protocol objects, **groupAssigner** and **formatter**. They both localize important design policies of the Persistence Framework. As a result, wide variation of the framework is possible by refining these encapsulated policies.

The **groupAssigner** uses a policy by which an object is assigned to the same IOGroup as its container (called parent in ObjChart terminology). This group assigner needs to mediate between the object to be stored (to find its parent's IOGroup) and the storage manager. The purpose of using a separate object is to encapsulate the policy, rather than burying it in the store method of **storageManager**, such that this policy can be independently refined should the reuser so desire. For example, in case of a quick word indexing over compound documents, perhaps it is logical to store all the word objects starting with a given prefix in a single IOGroup instead of storing them together with the paragraph objects (i.e., their containers). The object **groupAssigner** is an example of a so-called collaboration object, mediating between **storageManager** and the object to be stored.

Similarly, the linearization policy is encapsulated in another collaboration object, called **formatter**. It looks up from a given object the data to be stored, linearizes it according to its own algorithm, and passes the linearized data to be written to a specific IOGroup. In fact, it leaves the linearization algorithm itself as a virtual operation to be overridden by its concrete subclasses. The formatter participates in two interactions; it receives a *formatObj* message from some IOGroup for a specific object, in response to which it sends a message to the object requesting its attributes and values that need to be stored. Whenever an object sends a message with attributes and values to a formatter, the formatter linearizes the collection using its own algorithm (by invoking its operation named *doFormatting*) and sends the linearized stream back to the concerned IOGroup.

4. Conclusion and future work

In this paper we have argued that understanding the architecture is the key issue in reusing frameworks. We have illustrated why class retrieval is, at best, a minor problem. We have proposed that architecture can be better understood through interactive and tangible exploration of an exemplar. This further enables the reuser to create a new application by top-down customization/refinement of an exemplar, rather than searching through a vast collection of fine-grained classes and composing them bottom-up.

In this paper we have outlined the "Design by Framework Completion" approach for effective reuse of frameworks. There is a need to apply these techniques to different framework libraries, to ensure that active exploration can indeed be used as an effective means for understanding the architecture of a framework. Moreover, we foresee creating effective exemplars as akin to creating good tutorials, and thus will require multi-disciplinary cooperation among framework developers, technical writers and the human factors people.

The essential message of our work is that top-down completion of an instantiated architecture by (replacing and/or adapting) the available components is the only way to effectively reuse software. Following (Gangopadhyay, 1991), we have argued that the main problem of bottom-up component assembly is the incompatibility of architectural assumptions. Similar observation has been reported independently by Garlan et al. (1995), even when they had used fairly large-grain components. However, their recommended suggestions are aimed at alleviating architecture mismatches, still within the style of bottom-up component assembly. In contrast, Design by Framework Completion is a complete paradigm shift, by using architectural information as the foundation for effecting reuse, rather than to start with components and find if their architectural assumptions match. Not only do we represent architecture, but we ensure that only architecturally compatible components (chosen from the inheritance lattice) are used for completion. Moreover, by executing exemplars, we provide an intuitive approach of learning the architecture by active visual exploration, a form of "learning by example".

Our research has been performed in the context of object-oriented frameworks, because of their wider availability. However, we believe that our findings and the approach outlined here are equally well suited to reuse of non-OO components. Essentially, each class has the same property as an encapsulated module (or component). The inheritance lattice, which is an organization of classes based on commonality of semantics, can be viewed as a special case of component classification schemes. In our reuse receipe, the only step with a dependency on object-oriented frameworks is in the one entitled "finding alternatives", wherein we have used the inheritance lattice, which could just as well be replaced by any classification scheme mentioned before. We believe that such cross-fertilization of findings from the object-oriented community to general software reuse community will advance the state-of-the-art.

Appendix: ObjChart Constructs

For completeness, we present an overview of the modeling constructs of ObjChart. The visual notations for the ObjChart constructs are depicted in figure 10. Notice that this is just a brief exposition; more details (for example, on the semantics, object structure, finite state machines, etc.) appear in (Gangopadhyay and Mitra, 1993).

The central concept is that of an *object*. An object may be an instance of some *class*. Classes implement *types* for objects. Objects can have *attributes* (data members), *operations* and a collection of *ports*. Ports of objects are place-holders for potential collaborators; collaborators can be connected to ports using *connectors*. Objects in ObjChart may be composite; i.e., structurally, objects can be refined by adding subobjects. For example, **object** is a subobject of **composite** in the discussion on Persistence Framework.

Behavioral descriptions of objects are provided using causal rules. A causal rule consists of Event-Conditions-Actions tripple and defines a conditional response to an input stimulus. The *event* is the stimulus, which causes the object to perform the *actions*, provided the *conditions* hold. It may be convenient to partition causal rules based on enabling states, in which case a finite state machine (FSM) is associated with an object. Usually, behavior is initially defined abstractly using simple unconditional causal rules that define cross-object interactions (i.e., the Object Interaction diagram). At a later stage, these causal rules are further refined (say, by adding enabling state information, conditions, etc.) to provide precise behavior for objects.

Over and above the simple structural constructs discussed so far, ObjChart allows protocols to be encapsulated as objects, thereby enabling the treatment of protocols as first-class

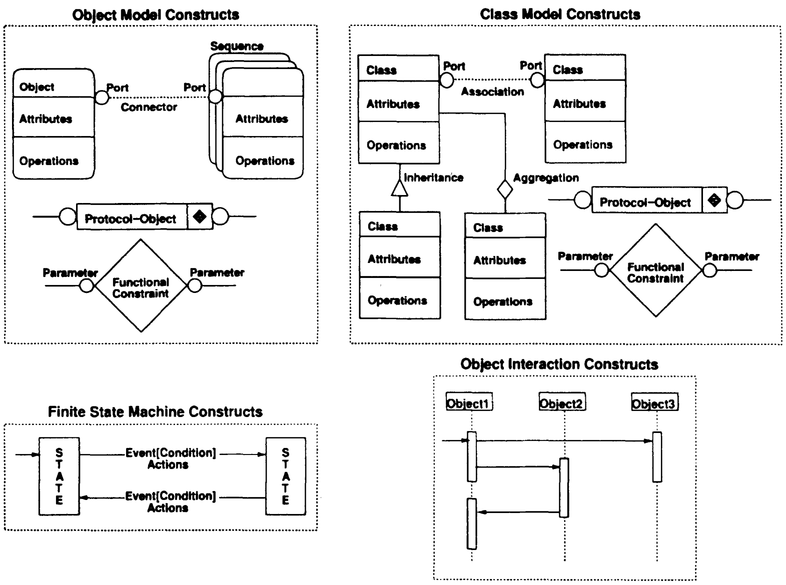


Figure 10. Constructs of ObjChart.

entities (for example, the same structural and behavioral refinement principles can be applied to protocols; also, protocols can be extended by adapting their classes, as done in the Abstract Factory example). Protocols may be viewed as reusable entities, in which the fixed part of a behavioral interaction is encapsulated in the protocol object, while the variable parts are abstracted out using ports. Causal rules of the protocol object describes the progress of the protocol in response to receipt of messages through its ports.

Finally, ObjChart has the *Sequence* construct which depicts an ordered collection of elements of a given type.

The ObjChart-Builder (Gangopadhyay et al., 1993) allows interactive exploration of ObjChart models. During execution, behavioral information stored in objects, by way of causal rules, is used to produce a trace-diagram, which shows the actual messaging interactions that occurred during the execution of the model. Such a trace, for the Persistence Framework, has been discussed before, and shown in figure 5.

5. Acknowledgments

We would like to thank Oliver Horn of IBM Germany, and the referees for their comments.

Notes

1. Unless otherwise stated, we would use the word framework to mean an object-oriented framework.
2. The example framework has been adapted from the SOM Framework Library (IBM Publication No S246-0108-00).
3. Note that the contained objects are not visible outside the container in the ObjChart representation. Thus, the sequence **persistentObjects** is not visible at this level of the model. Similarly **object**, which is contained within **composite**, is also not visible. Such hiding of the details is essential for examining a model abstractly. In case further details is required, it is possible to descend into the structure by opening any container object, at which point the contained objects become visible (see the Appendix and (Gangopadhyay and Mitra, 1993) for details).
4. Notice that these scenario traces are generated from the behavior definition of the objects, specified by either causal rules or FSMs; see the Appendix for further details.
5. The objects in figure 3 are instances of the classes shown in figure 7; for example, **composite** is an instance of **compositeClass**, **groupAssigner** of **grAssgnByParClass**, **storageManager** is an instance of some concrete subclass of **storageMrgAbstract**, which we have omitted for brevity, and so forth.

References

De Pauw, W., Helm, R., Kimelman, D., and Vlissides, J. 1993. Visualizing the behavior of object-oriented systems. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA) 1993*, ACM SIGPLAN Notices, 28(10):326–337.

Deutsch, L.P. 1989. Design reuse and frameworks in the smalltalk-80 system. T.J. Biggerstaff and A.J. Perlis (Eds.), In *Software Reusability*, ACM Press, pp. 57–72.

Fischer, G. 1987. Cognitive view of reuse and redesign. *IEEE Software*, pp. 60–72.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1994. Design patterns: Elements of reusable object-oriented software, Addison-Wesley.

Gangopadhyay, D. and Helm, A.R. 1989. A model driven approach for the reuse of classes from domain specific object-oriented class repositories. *Research Report RC14510*, IBM Research Division.

Gangopadhyay, D., Mitra, S., and Dhaliwal, S.S. 1993. ObjChart-builder: An environment for executing visual object models. In *Proceedings of the Eleventh International Conference and Exhibition of TOOLS USA* (Technology of Object-Oriented Languages and Systems).

Gangopadhyay, D. and Mitra, S. 1993. ObjChart: Tangible specification of reactive object behavior. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP'93)*, 707:432–457 of Lecture Notes in Computer Science.

Gangopadhyay, D. 1991. On tangible representation of compositions of software components. Position paper. In *Proceedings of 14th Minnowbrook Workshop on Software Engineering.*

Gangopadhyay, D. 1994. ObjChart User Guide, IBM Internal Report.

Garlan, D., 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 in Software Engineering (ICSE-17).*

Goldberg, A. and Robson, D. 1984. SMALLTALK-80: The Language and Its Implementation, Addison-Wesley, Chap. 17.

Helm, R., Holland, I.M., and Gangopadhyay, D. 1990. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of OOPSLA/ECOOP'90*, ACM SIGPLAN Notices, 25(10):169–180.

Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G. 1992. Object-oriented software engineering: A use case driven approach, Addison-Wesley.

Maarek, Y.S. 1990. Indexing software components for reuse by using natural language documentation. In *Proceedings of the Third Annual Workshop: Methods and Tools for Reuse.*

Pietro-Diaz, R. and Freeman, P. 1987. Classifying software for reusability. *IEEE Software.*

Pree, W. 1994. *Designing with Patterns*, Addison-Wesley Publication.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. 1991. Object-oriented modeling and design, Prentice-Hall.

Rollins, E.J. and Wing, J.M. 1991. Specifications as search keys for software libraries. In *Proceedings of 8th International Conference Logic Programming.*

Rosson, M.B. and Carroll, J.M. Active programming strategies in reuse. In *Proceedings of the European Conference on Object Oriented Programming* (*ECOOP'93*), 707:4–20 of Lecture Notes in Computer Science.

Soetarman, B. 1994. Personal Communication.

Schappert, A. 1994. Demo at OOPSLA.

Introduction to OS/2 System Object Model, IBM Publication No. S246-0108-00.

Commonpoint Application System: Documented Samples, Publication No. Z123-7461-0, 1995.

Automated Software Engineering 3, 239–259 (1996)
© 1996 Kluwer Academic Publishers. Manufactured in The Netherlands.

Building an Organization-Specific Infrastructure to Support CASE Tools

SCOTT HENNINGER scotth@cse.unl.edu
Department of Computer Science & Engineering, University of Nebraska-Lincoln, 115 Ferguson Hall, CC 0115

Abstract. CASE tools are notorious for forcing organizations to adapt to a standard development methodology. The underlying assumption is that a universally applicable development method exists and it is up to the organization to conform to that method. But software development is no longer a homogeneous field. As computers are applied to an increasingly diverse set of applications, it is becoming increasingly important to understand the different demands these domains place on the development process. Our solution to this dilemma is to create an organization-wide development infrastructure based on accumulated experiences within application and technical domains. The domain lifecycle formalizes a process for accumulating project experiences and creating domain knowledge than can be used to increase product quality and improve development productivity. Supporting the domain lifecycle eases development of software that has been developed previously in the organization, freeing designers to concentrate on less well-known elements of an application.

Keywords: domain lifecycle, organizational learning, methodology, reuse, domain analysis, CASE adoption, groupware

1. Adapting an organization to CASE tools

The promise of the CASE industry has been to improve software productivity and quality through technological innovations aimed at automating parts of the development process. While many have touted CASE technology as the tool that will revolutionize the software development industry (Yourdon, 1992), these claims remain largely unfounded. The few studies that have been attempted are either equivocal or contradictory, with some showing a perception among developers that CASE improves productivity and quality (Norman and Nunamaker, 1989), others showing some improvement in productivity and quality for medium-scale programs (Granger and Pick, 1991), and others showing little or no effect on productivity and quality (Card et al., 1987). Studies have shown that introducing CASE tools to an organization is tenuous at best, with organizations choosing to adopt CASE tools in limited forms and abandoning much of the technology soon after its introduction (Kemerer, 1992).

Part of the reason for this confusing state of affairs is that, in spite of the desire to create universally applicable solutions, the same CASE tool applied to different development contexts or different development organizations may yield entirely different results. CASE tools are often coupled with specific development strategies and are often ill-suited for development under different strategies (Vesssy et al., 1992). For example, a CASE tool supporting a traditional waterfall model might prohibit developers from going into design and development stages before specification or requirement phases have been certified as "finished" (Ramanathan and Sarkar, 1988). This will not work well in projects needing one or more prototypes to flesh out system requirements and assess design risks (Boehm, 1988). While other design disciplines have come to recognize that proper tools must be used for the job, research in the software engineering and CASE communities continue to search for universal solutions to a complex and multi-faceted problem characterized by different application domains (Curtis et al., 1988), various design and implementation strategies, numerous organizational styles, and user demands for high-level domain-specific and easy-to-use software that achieves high levels of portability, modularity, and robustness at increased cost effectiveness (Urban and Bobbie, 1994).

But universally applicable solutions do not exist, and organizations wishing to make an investment in CASE are faced with a multi-dimensional assessment task. Not only must an organization assess the quality of the CASE tools, it must also determine how well the tool integrates with the existing computational environment and corporate infrastructure, how well it supports specific development strategies, and the expressive power of the CASE tools for the kinds of problems it will be used to solve. Hidden costs in the form of training and learning to use the tools effectively, a process that can consume up to twice the actual purchase price of a CASE tool (Huff, 1992), can also have unforeseen and undesirable effects. CASE tools that are incompatible with corporate development strategies and organizations will mandate sweeping changes to the process of managing and developing software. These tradeoffs need to be identified and analyzed to find the most effective CASE tool for a given organization and types of development projects and strategies used by the organization.

1.1. Adapting CASE tools to an organization's development context

The approaches to integrating CASE tools into a development organization have essentially followed two paths. The first is to impose a set of functionality and a prescribed development process and let the organization adapt itself in order to make effective use of the tools. While this eases the burden on CASE tool developers, it is often difficult to radically change development practices in an organization, leading to CASE shelfware and adoption failures (Kemerer, 1992). This solution also assumes that there exists a solution to "the" software engineering problem (and the CASE vendor will argue hard that they've found that solution) and it is simply a matter of getting organizations to adopt their practices accordingly. The second approach is to provide a set of features that can be configured to meet a development organization's needs. This adds a necessary degree of flexibility, but still assumes that there exist variants of a universal solution. To the extent that an organization's development needs fit these pigeon holes, the tool is successful. But the dismal reports of CASE in practice suggest that the pigeon holes don't exist or at least the right ones haven't been discovered.

The learning curve has long been recognized as a barrier to adoption (Kemerer, 1992). Yet the CASE field is often criticized for only providing point solutions. A recent study showed that out of 14 tools studied, most provided generation of header files, only 2 provided full code generation, most had weak forms of consistency checking, none fully automated document generation, and tools that were strong in one area were often weak in others (Church and Matthews, 1995). If CASE tools mostly provide point solutions (such as requirements engineering, database generation, etc.) and face a steep learning curve, then it stands to reason that comprehensive solutions are even more complex with larger learning curves.

The problem is that viewed in the abstract sense, software development (or software engineering, if one prefers) is an intractable problem. While a single comprehensive CASE solution would certainly be convenient, diversity is necessary to meet the needs of software development in disparate application domains and technologies. Any single system meeting all these diverse needs is beyond our capability to develop or understand software systems. Viewed in this manner, point solutions will continue to be offered by CASE vendors for some time to come. The means are therefore needed to manage the point solutions effectively to ensure product quality throughout the development lifecycle.

Adopting CASE technology is therefore more complex than enforcing organizational compliance to a methodology or providing customization tools. Information is needed on *when* a CASE tool should be used, and *how* it should be employed to solve a specific problem. This information is inherently organization-specific, as it involves the entire development context, including expertise of the developers, application domains, hardware used by developers and customers, organizational culture, and many other factors.

In this paper we propose an alternative to adopting CASE technology. As opposed to adopting an organization to CASE solutions or providing superficial customizations, development organizations create an infrastructure within which CASE tools can be used effectively. This approach involves the development and evolution of organization-specific knowledge about how CASE tools are used most effectively within the organization's development context. Feedback from developers is used to extend CASE tools to meet the specialized needs of the organization and help developers understand how the tools can be used to solve specific problems. This approach recognizes the growing diversity of user populations and the corresponding demands this diversity places on the development process (Henninger and Lappala, 1994). There is no magical potion, no silver bullet, that can be simply purchased to solve all development problems in a given organization. In the same way that the Software Engineering Institute's Capability Maturity Model (CMM) (Humphrey, 1989) advocates a continual improvement process, our approach advocates the development of an infrastructure that is used to identify frequently occurring problem areas where an organization can benefit from efforts to automate development within the domain.

2. An organizational framework for CASE tools

Existing CASE tools and existing development methodologies focus exclusively on the project lifecycle with only incidental reference to previous development efforts, development infrastructure, and other organizational factors affecting the development process. Methods are needed that codifies knowledge as it emerges in development organizations into a form that can help make developing routine software more routine (Computer Science and Technology Board, 1990). This knowledge is treated as a corporate asset that is used as a basis for continual improvement of software products and the development process. Software reuse is practiced throughout the development process through a methodology in which design decisions are based on prior experiences. Software reuse and domain analysis become the *basis* for designing software systems, not just an implementation technology.

Appropriate levels of formality is supported as an organization learns about the domains their software is built around. We address these needs in a progression from individual cases to domain-oriented design environments that formalize development knowledge and artifacts. This progression defines a *domain lifecycle* (Simos, 1988) in which the focus is shifted from individual projects to recurring design problems within an organization.

2.1. An organizational learning approach to software development

An organizational learning approach to software development addresses these issues by capturing project-related information during the creation of individual software products (Henninger et al., 1995). This information can then be disseminated to subsequent projects or domain analysis efforts to provide experience-based knowledge of development issues encountered at the organization. Given a repository of experiences with tools and methods, new projects can match their characteristics to the characteristics of existing solutions. These solutions provide knowledge about how successful a given method or tool was for that kind of domain, provide how-to information for specific problems, identify potential problems, etc. Information in the repository not only points to re-usable solutions and code, but also suggests how novel problems can be approached by adapting similar problems, warn of possible problems with solution methods, and help designers understand the boundaries of a problem.

An organizational learning approach to software development uses an organization's accumulated knowledge of the development process and application domains as the basis for design. While the ultimate goal may be to develop automatic programming tools (Rich and Waters, 1988), coalescing and analyzing the necessary knowledge to achieve this goal is a difficult process that can only be accomplished in well-understood domains (Biggerstaff, 1992; Prieto-Díaz, 1991). Intermediate methods are needed that can disseminate knowledge as it is created in the organization so people can begin to build a culture based on success, avoid duplicate efforts, and avoid repeating mistakes. These techniques provide information relevant to local development practices that "you can't learn in school" (Terveen et al., 1993), such as custom languages, organization and project-specific programming conventions, policies and guidelines concerning tool usage, development standards, individuals with expertise in specific areas, and many others.

By this definition, domain knowledge does not necessarily need to be centered around a common family of applications or a formal model (Arango, 1989), but a set of problems within applications with recurring activities and/or work products. In fact, the most valuable kind of knowledge can be characterized as "lessons learned"; mistakes and sub-optimal solutions that developers don't want to repeat, tips and techniques for accomplishing specific tasks, and methods that proved to be successful. The development process in general can be refined and streamlined by identifying commonly occurring patterns, thereby reducing the amount of duplicated effort and learning from the mistakes and successes of projects with similar characteristics. The emphasis becomes one of systematically institutionalizing knowledge as it is generated by people in the organization.

As the cases accumulate in the organization-wide repository, the knowledge contained in the repository becomes increasingly tailored to the kinds of design problems that frequently

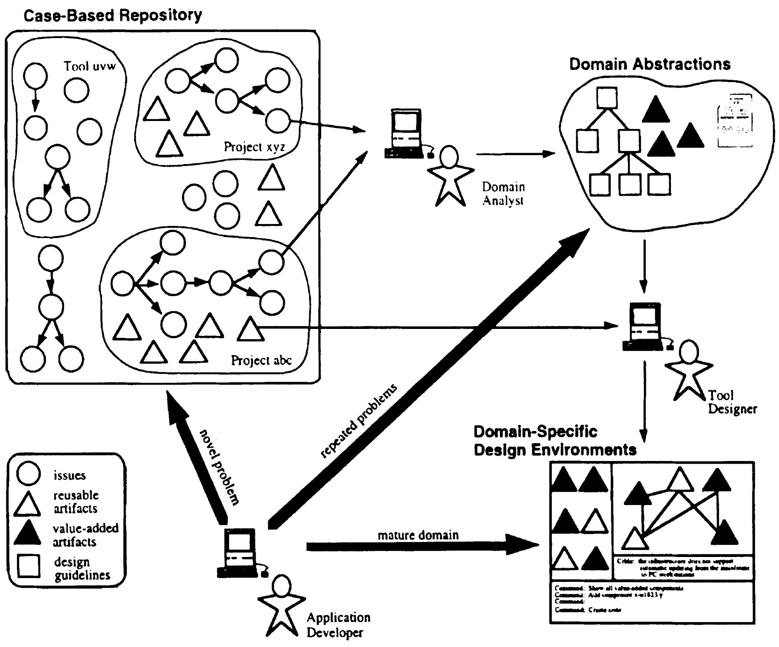


Figure 1. Organizational learning support for the domain lifecycle.

occur in the organization. The repository therefore serves not only as a means to disseminate design knowledge, but also helps an organization *learn* what does and does not work for their development context. This is where we distinguish between organizational *memory* systems that many have advocated (Walsh and Ungson, 1991) and our notion of organizational *learning*, where the emphasis is placed on learning from previous experiences. The method also naturally incorporates an evolutionary, continuous, process of improvement that evolves with the ever-changing development context. As developers in the organization gain experience with problems in the domain, the domain evolves toward more formal and higher quality representations, defining a lifecycle for problems with similar characteristics.

Figure 1 depicts how an organizational learning approach to software development supports the domain lifecycle. As frequently encountered problem domains mature from novel problems to repeated problems to a fully mature domain, support is provided in increasing levels of automation. CASE tools supporting the three main steps in the domain lifecycle are identified:

- A **Case-based repository** collects experiences from individual projects, tool experiences, and other artifacts generated to support the development infrastructure of the organization. Project experiences and design artifacts are collected through status reports, project management, and design rationale that describes the problems that are addressed while creating an application. Tool experiences are how-to advice and descriptions of problems encountered while developers are using a tool to develop software. This provides solutions to organization-specific problems that are not found in manuals. Reusable artifacts can be in the form of procedures for approaching a problem (process models), software modules, specifications, frameworks and architectures, requirement documents, algorithms, designs, test suite, and other items generated in the course of a project.

- **Domain abstractions** are domain-specific models of design problems, including design guidelines, value-added reusable artifacts, domain-specific handbooks, process models, design checklists, and other forms of knowledge. Domain-specific knowledge is created by a domain analyst refining knowledge contained in the case-based repository into forms that are more generally applicable to problems frequently encountered in the organization.
- **Domain specific design environments** automate or provide knowledge-based support for the development of systems within well-established domains. The environments are created by tool designers using accumulated knowledge from the domain models and reusable artifacts in the case-based repository.

These steps define increasing levels of automation and support mirroring the maturity of the domains. As shown with the heavy arrows in figure 1, novel problems are supported by searching for similar problems in a case-based repository of project and tool experiences. The cases will contain information that is specific to the original project and may need work to apply to the current context, but at least there are some prior experiences to help guide design decisions. As activities are repeated, case-based technology is employed to identify recurring development issues and support the process of generalizing from individual cases to domain-specific abstractions. Using handbooks, guidelines, and domain models provides a higher level of support because the knowledge has been processed by domain analysts into a form that is applicable to general problems in the domain. As the domain matures, tool designers can use the synthesized knowledge and components to construct design environments that automate design and provide intelligent support for mature domains repeatedly encountered in the organization.

2.2. *Empirical work on organizational memory*

We are currently engaged in a project that uses the "industry-as-laboratory" (Potts, 1993) approach to explore the domain lifecycle in a technically sophisticated in-house software development organization at Union Pacific Railroad (UPRR) with about 350 software developers. We have systematically studied software development at UPRR with structured interviews (Curtis et al., 1988), contextual inquiries (Holtzblatt and Jones, 1993), and diary studies (Rieman, 1993). Through these studies we have collected extensive notes and hours of video recorded information with an even mixture of developers and project managers.

Our foremost conclusion from these studies is that a combination of diverse development concerns, complexity and novelty in the development environment, and many relatively small-scale individual projects are working together to exacerbate the thin spread of application domain knowledge (Curtis et al., 1988). There are currently 139 separate projects (59 of which are new development) in 12 different functional areas of the business, ranging from order processing and revenue management to dispatch monitoring, resource planning, and scheduling. In addition to intra-project communication needs, there is a need for communication between these projects, as they share concerns of the application domain (aspects of the railroad business) as well as common development platforms. Current organizational lines tend to create barriers for this kind of communication (Poltrock and Grudin, 1994), creating a lack of consistency across products and duplication of effort.

For example, in a design meeting we observed at UPRR, a team was building an application that moved a dataset from the mainframe to the more accessible medium of Lotus Notes on PC workstations. The program was designed to be manually invoked from a workstation. This aspect of the design caused a great deal of discussion about the merits of automatically triggering the program from the workstation:

S1: "So then, my question becomes: Are we not far enough in our infrastructure where we can automatically trigger this job and move it over to Lotus Notes?"

Automatic job triggering is clearly an issue with broad applicability in UPRR's client-server architecture, for which no known solution exists, but similar issues have been addressed:

S2: "Currently we have jobs that automatically run and populate... be it a Oracle database or whatever!"
S3: "On the server!"
S2: "Right, going through server. But I don't know about Lotus Notes, so I can't speak of that! But I know there are things out there that could potentially do that. They can do that through client-server 10–33 machines. So it should be able to do that in Lotus Notes Database. I know there are other systems out there that are working to get information directly from Oracle, which is only 10–33 machines, and again leading it into Lotus Notes. So there is a potential option and that is to move it straight to Oracle and then port it out of Oracle."
S3: "Yeah, I think, in fact, there are lot of tools available today and working to automatically initiate jobs on the server. But this one is kind of unique in that it has to be initiated from the workstation. But I don't know whether there are any to remotely initiate jobs on the workstation."

Here we have a pattern, a recurring need for the organization that needs to be identified and disseminated to projects potentially needing a solution. Its utility may seem obvious in hindsight, but it is an emergent need to the people in this organization. As this project finds a solution, it needs to be disseminated so projects with similar requirements can use, adopt, and learn from the solution. Methods are needed that disseminate what is currently known in a timely manner so the organization can build on successes, avoid duplicate efforts, and avoid repeating mistakes.

The following sections present some examples of how CASE tools can support the domain lifecycle. The maturity of these examples vary, reflecting our current state in this multi-year project. We are currently in the process of installing tools for the case-based repository in a pilot project at UPRR. The purpose of presenting them in this paper is to demonstrate how the domain lifecycle can be supported, not to present final versions of CASE tools.

3. Supporting the domain lifecycle with organizational learning tools

An organizational learning approach to software development requires the development of three interrelated techniques:

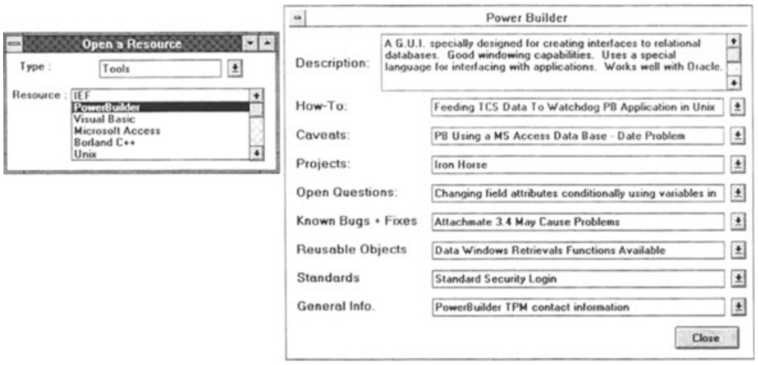
- case-based methods to collect and organize project experiences, tools, and development infrastructure issues
- tools to support the identification of domains and synthesize knowledge in the form of domain models, design guidelines, value-added reusable artifacts, and electronic handbooks
- development of domain-specific design environments to provide automation and intelligent support for well-established domains.

3.1. A case-based software development repository

An organizational learning approach to software development depends critically on identifying commonly occurring patterns across a number of development efforts. Case-based reasoning is an artificial intelligence method based on cognitive models postulating that much of human problem solving involves applying past experiences to analogically related situations. While early case-based systems attempted to provide artificially intelligent problem solving by automatically adapting existing solutions to new situations, recent systems have emphasized providing an external memory for users through an interactive process of decision support (Kolodner, 1991). A case-based repository for decision support helps users reason "from old cases or experiences in an effort to solve problems, critique solutions, explain anomalous situations, or interpret situations". (Kolodner, 1991; p. 53).

Case-based decision aid technology is a perfect fit with an organizational memory approach to software development because we are interested in situations in which no formalized or algorithmic solutions are available, but problem solving examples exist. Users begin the problem solving process by describing a problem to the case-based system. The system retrieves cases with similar features. Using these cases as a basis for decision making, the user adapts case solutions to their problem solving context. The cases help the user by showing suggestions for solutions, problems encountered in old cases, and results of applying the case (Domeshek and Kolodner, 1992). Even when the context in question is radically different than the available cases, the information can help designers focus on issues that significantly impact the design process and artifact. Case-based methods can also support the abstraction process that is so important to domain analysis (Prieto-Díaz and Arango, 1991) through the detection of patterns, such as when several cases suggest the same solution and/or are indexed with similar terms.

Using case-based methods for constructing a repository of software development knowledge involves three interrelated issues: (1) effective knowledge dissemination, (2) unobtrusive knowledge collection, and (3) changes to the development process. In what follows, these issues are examined through a second-generation prototype that we have developed to communicate and explore our ideas with UPRR personnel.


Open a Resource

Type : Tools

Resource : IEF
PowerBuilder
Visual Basic
Microsoft Access
Borland C++
Unix

Power Builder

Description:	A G.U.I. specially designed for creating interfaces to relational databases. Good windowing capabilities. Uses a special language for interfacing with applications. Works well with Oracle.
How-To:	Feeding TCS Data To Watchdog PB Application in Unix
Caveats:	PB Using a MS Access Data Base - Date Problem
Projects:	Iron Horse
Open Questions:	Changing field attributes conditionally using variables in
Known Bugs + Fixes	Attachmate 3.4 May Cause Problems
Reusable Objects	Data Windows Retrievals Functions Available
Standards	Standard Security Login
General Info.	PowerBuilder TPM contact information

Figure 2. Viewing information on tools.

3.1.1. Knowledge dissemination. With over 90 different development tools in use at UPRR, choosing an appropriate set of tools for a project is becoming a significant problem. The sheer number is a formidable barrier, but the complexity and overlapping nature of these tools, ranging from operating systems, databases, and languages to CASE tools, development methodologies and word processors creates an information overload situation. Our approach to disseminating case-based knowledge has been to provide CASE tools to support exploring the repository (Henninger et al., 1995). Two methods are provided for retrieving and browsing information in the repository. The first is to find a specific tool and search its categories for needed information. The second is to search across all cases, receiving matching cases the cross tool boundaries. The methods are described in the following paragraphs.

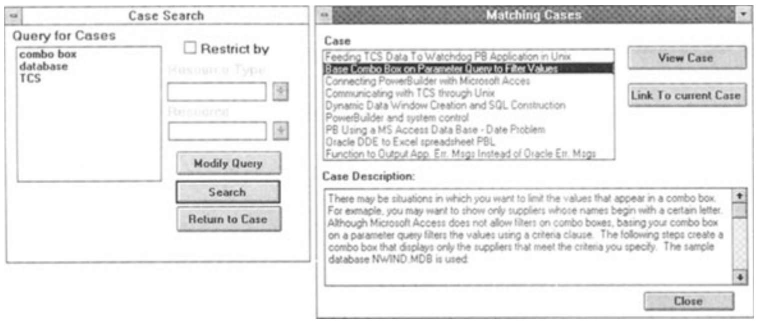
Finding tool information. Figure 2 demonstrates how information about resources are viewed. Resource windows are one-stop locations to get information on a specific CASE tool, development method, etc. The "View a Resource" window on the left-hand side of figure 2 is used to choose a type of resource, which includes tools, development methods, project issues, source code, and projects. Once the type is chosen the user can choose a specific resource from the scrollable window. Recent feedback has indicated that, with the number of available tools, another level of categorization may be necessary to make the process of finding the tool easier. Double-clicking on the resource name displays the window on the right-hand side of figure, in this case the PowerBuilder CASE tool.

The categories displayed in the resource interface (PowerBuilder in figure 2) is largely dependent on the type of resource. The fields we have chosen for tools were derived from a number of Lotus Notes databases used to exchange information about some of the tools used at UPRR. The Description field gives a brief introductory message about the tool. The How-To field contains tips and techniques that have been used to solve problems with the tool. The entries in this and the rest of the fields point to cases in the repository that can be displayed by clicking on the entry. Notice that the How-To entries contains organization-specific knowledge about how the tool is used in the development context at UPRR which includes an eclectic mix of other off-the-shelf CASE tools, in-house systems, projects, and other issues. For example, ‘TCS’ refers to the “Train Control System”, a custom-built mainframe application used to collect large amounts of data about the railroad, while ‘Watchdog’ refers to a project in the organization.

The Caveats field is intended to collect information about idiosyncratic issues users may need to be aware of when using the tool. For example, some UPRR projects have discovered that PowerBuilder’s control of the system can interfere with communications software. The Projects field collects all the UPRR projects that have used this tool. Open Questions is a forum for unresolved questions. Once an open question is answered, it should migrate to the How-To, Caveats, or Known Bugs and Fixes fields. The Known Bugs and Fixes field holds information about bugs with the tool or using the tool in certain environments. Reusable Objects points to information about reusable objects associated with the tool along with pointers to the source. Standards holds corporate standards for using the tool, naming standards, etc. General Info. helps users find people with expertise about the tool. It includes local experts (“TPM” refers to Technology Product Managers that are responsible for helping developers use applications effectively). It also has contact information for the CASE tool vendor.

Querying for similar problems. Developers may not always need information that is tied to a specific tool. In these cases we have provided users with a search mechanism that finds cases across different tools and projects. Figure 3 shows the query interface and the result of a query about combo boxes for access to TCS (Train Control System) databases. The matching cases shown in the window on the right-hand side of figure 3 display the titles of cases and the description field of the selected case. Double-clicking on the title of selecting and choosing “View Case” displays the case’s window (a case window is shown in figure 5). In this instance the user has chosen to search the entire repository of cases. By choosing the “Restrict By” box, the user can restrict the search to a particular type of resource and (optionally) a specific resource (such as PowerBuilder or IEF).

This query method is invaluable not only for answering specific queries, but for finding out what kinds of problems a given tool can be used for. By querying the repository for a given problem, cases are returned that match that kind of problem. By perusing what kinds of tools have been used to solve this problem, one can begin to understand which tool should



combo box
database
TCS

Feeding TCS Data To Watchdog PB Application in Unix
Base Combo Box on Parameter Query to Filter Values
Connecting PowerBuilder with Microsoft Acces
Communicating with TCS through Unix
Dynamic Data Window Creation and SQL Construction
PowerBuilder and system control
PB Using a MS Access Data Base - Date Problem
Oracle DDE to Excel spreadsheet PBL
Function to Output App. Err. Msgs Instead of Oracle Err. Msgs

There may be situations in which you want to limit the values that appear in a combo box. For exmaple, you may want to show only suppliers whose names begin with a certain letter. Although Microsoft Access does not allow filters on combo boxes, basing your combo box on a parameter query filters the values using a criteria clause. The following steps create a combo box that displays only the suppliers that meet the criteria you specify. The sample database NWIND.MDB is used.

Figure 3. Searching for cases.

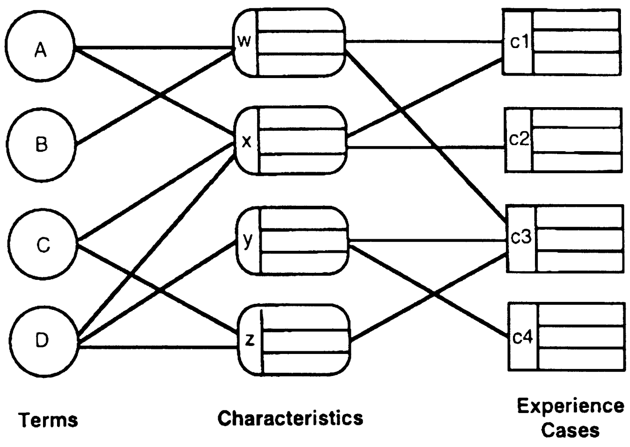


Figure 4. Indexing architecture for the repository.

be used for a particular type of problem. For example, it could be found that IEF has been used for a number of systems in which access to windows and buttons need to be secured for certain users. Using this tool for a development project with similar requirements ensures that some of the issues have already been documented, and increases the likelihood that reusable objects exist.

Any case-based approach relies heavily on the case retrieval mechanism, often referred to as the *indexing problem* (Kolodner, 1993), which is responsible for finding appropriate cases for a given problem description. Indexing, the process of representing cases with key terms and phrases (the "Characteristics" field in figure 5), is only half the problem. The other half is the method of matching queries to case representations. Simple matching techniques have been shown to inadequately support the process of satisfying an information need, especially when the query is ill-defined (Belkin and Croft, 1992). Methods are needed that can retrieve noisy and inexact patterns with a soft matching retrieval algorithm.

The indexing architecture we have adopted consists of three types of objects; terms, characteristics, and experience cases (see figure 4). People searching for experience cases specify a query with characteristics. Characteristics are structured objects with a description, a list of cases that use the characteristic, and a list of terms that index the characteristic. They define a standardized controlled vocabulary to index cases. People indexing cases are encouraged to reuse existing characteristics when they apply, although new characteristics can easily be defined. A controlled vocabulary approach was adopted for three reasons. First, for describing objects, such as source code, that do not follow the linguistic regularities of text documents, controlled vocabulary approaches may be superior to other indexing methods (Prieto-Díaz, 1991). Secondly, this approach fits many organizations where standard terminology and acronyms are used to communicate common issues. We often heard statements like "That's a track capacity issue" at UPRR. Key phrases such as "track capacity" can be used as characteristics to help establish a carefully designed vocabulary that best describes domains within the organization. Third, defining a standard set of terminology is a first step toward formalizing domain knowledge (Prieto-Díaz, 1991).

The problem with a standard or controlled vocabulary is that it must be learned. This is a barrier not only to novices, but experienced people that are exposed to new projects with their own set of terminology. We therefore allow an uncontrolled vocabulary of terms to help find characteristics, as shown in figure 4. People need to use characteristics to look for cases, but if they are unsure of which characteristics to use, or want to find an exhaustive list of characteristics for a given issue, they can construct a query of terms to find characteristics. We do not allow terms to retrieve cases as this would reduce the benefits of using a controlled vocabulary.

We have chosen a spreading activation retrieval method that uses a connectionist relaxation algorithm to support finding partially matching patterns. The algorithm is explained formally elsewhere (Henninger, 1995), but the basic process is as follows. Let's say a user specifies term A in a query (see figure 4). The A node is given an *activation value* of 1.0 that is passed to all characteristics it indexes, w and x in this case. The activation value passed to the characteristic nodes will be reduced by the strength of the link weight (which measures the degree of association between a term and characteristic), and is adjusted by other factors such as fan-in and decay (Henninger, 1995). On the next cycle, w and x will have a non-zero activation value that will be passed to all term nodes they are connected to. This process repeats until activation values stabilize or a user defined number of cycles is reached. The same process is used to find experiences cases with characteristics defining the query.

The strength of this method is that it is able to find partial patterns in the repository. For example, when x passes its activation value to C and D on the second cycle, these two nodes work together to reinforce x's activation value and activate z. Further cycles reinforce x and z because of the feedback loop between these nodes and C and D. In the end, x and z will have similar activation values. The structure of the repository detects that characteristics x and z are similar because they have similar representations. The spreading activation process has detected the pattern through a partial match. Notice also that z would not have been retrieved if we were using a straightforward matching algorithm. Spreading activation found z because it is similar to x, which directly matched the query of A. While other partial match paradigms, such as Latent Semantic Indexing (Deerwester et al., 1990) and Lexical Affinity (Maarek et al., 1991), can also find partial matches, the spreading activation method was chosen because it is particularly suited to retrieving non-text objects such as source code (Henninger, 1994).

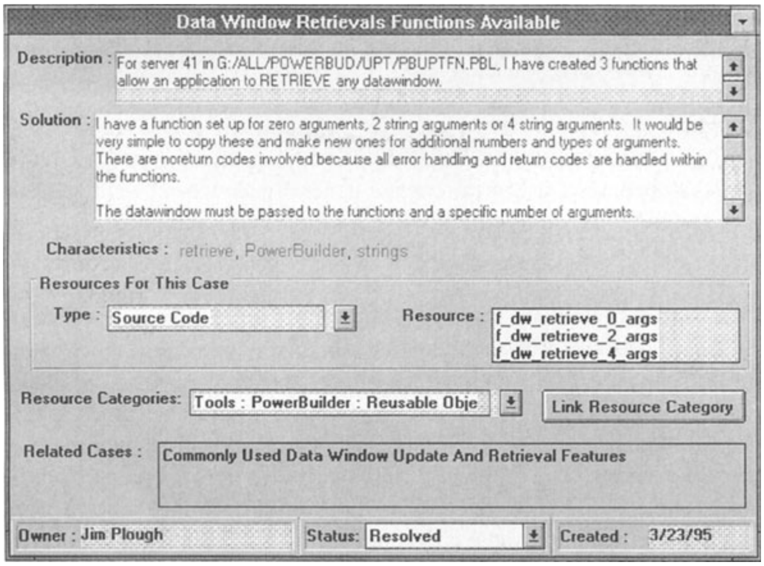
3.1.2. Knowledge collection. Developing a large-scale, real-world, knowledge base to support the development process is not a simple matter of engaging in an up-front knowledge acquisition effort. A process of continuous incremental refinement (Terveen et al., 1993) much be in place so the repository can evolve as new problems and solutions are discovered (Gaines, 1989). Generally there are two approaches to knowledge acquisition in software design environments. The first is to take the naive position that designers will readily perceive the future benefits of engaging in the often difficult, and always extra, task of putting knowledge into the system. A second approach is to appoint a "knowledge librarian" to maintain a knowledge base. While this shifts the workload away from developers and ensures that the knowledge base is continually updated, it does not ensure that relevant and accurate information is acquired.

This brings us to a third approach that provides knowledge acquisition tools so developers can encode knowledge as part of their actual development work. This leads to *contextualized* knowledge acquisition where knowledge is encoded into the system when it is created. Based on the premise that you won't know what is really needed until you're *in* the design process, contextualizing the knowledge acquisition process helps ensure that relevant knowledge is put in the knowledge base. This approach hasn't been explored much, but the underlying framework is already in place with domain analysis and design rationale techniques.

Our studies revealed that there are currently many information collection activities in development practices, with developers and managers at UPRR devoting an about of 30–40% of their time to knowledge collection activities such as entering tips and techniques into Lotus Notes databases, writing up meeting notes, and coordinating development activities. Scribes are often appointed at meetings to take notes and keep track of action items. We are currently in the process of putting some of these knowledge collection sources together and designing project management tools that capture design rationale throughout the design process for storage as cases in the repository.

One of the primary knowledge collection tools in the case window (see figure 5). This window contains all of the relevant information about a case, including a description of the problem, the solution, a set of characteristics (index terms), the resources associated with this case, resource categories, a set of related cases, and some status information on the case. All of these fields are optional and can be filled in either by creating a new case or modifying an existing case and saving it with a new name.

The case shown in figure 5 directs users toward some reusable source code that has been developed for the PowerBuilder platform. The case should therefore be associated with the


Data Window Retrievals Functions Available

Description : For server 41 in G:/ALL/POWERBUD/UPT/PBUPTFN.PBL, I have created 3 functions that allow an application to RETRIEVE any datawindow.

Solution : I have a function set up for zero arguments, 2 string arguments or 4 string arguments. It would be very simple to copy these and make new ones for additional numbers and types of arguments. There are noreturn codes involved because all error handling and return codes are handled within the functions.

The datawindow must be passed to the functions and a specific number of arguments.

Characteristics : retrieve, PowerBuilder, strings

Resources For This Case

Type : Source Code

Resource : f_dw_retrieve_0_args
f_dw_retrieve_2_args
f_dw_retrieve_4_args

Resource Categories: Tools : PowerBuilder : Reusable Obje

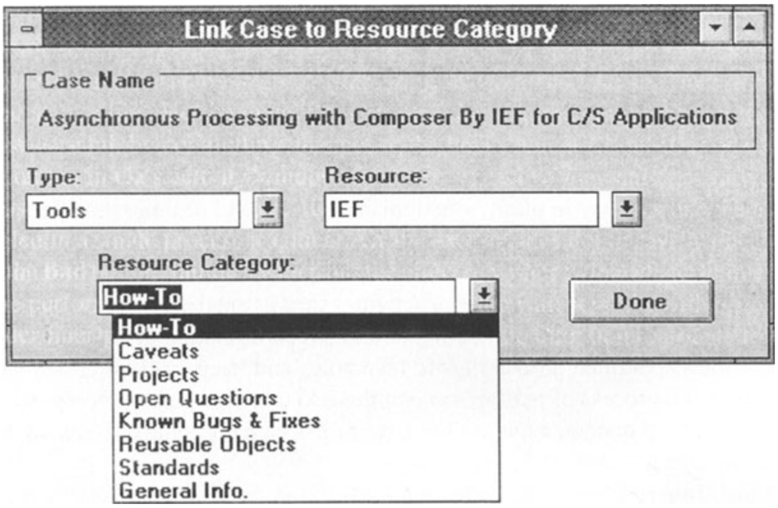
Related Cases : Commonly Used Data Window Update And Retrieval Features

Owner : Jim Plough

Status: Resolved

Created : 3/23/95

Figure 5. The case window.


Link Case to Resource Category

Case Name
Asynchronous Processing with Composer By IEF for C/S Applications

Type: Tools

Resource: IEF

Resource Category: How-To

Figure 6. Linking cases to a resource category.

“Reusable Objects” category of the PowerBuilder tool. This association is created by linking the case to the appropriate resource. Cases can be associated to any number of categories. Clicking on the “Link Resource Category” button brings up the window shown in figure 6. We used a different case for this image that has to do with asynchronous processing with the IEF tool. After clicking on the “Done” field, the case will be come associated with the “How-To” category of the IEF tool. The case will appear on the “Resource Categories” combo box in the case’s window and can be chosen from the “How-To” field in the IEF resource descriptor window (similar to the window shown in figure 2).

3.1.3. Changes to development practice. By enforcing the use of standard development methodologies and processes, most CASE tools mandate changes in the manner in which software is developed. But organizations are generally slow to change, causing a significant barrier to CASE tool adoption. The organizational learning approach to software development faces similar problems. In the very least, development becomes a process of basing decisions on previous experiences, thus advocating reuse throughout the development process. Institutionalizing reusability procedures in organizations is currently a hot topic, and few instances of true organizational change have been reported (Griss, 1993). The domain lifecycle itself advocates viewing software development as producing *product lines* (Gomaa and Kerschberg, 1995) with implications across the organization. This is a sizable departure from the current software development culture, which focuses exclusively on development *projects*.

The strength of the domain lifecycle is that proper levels of formalization can be applied as a domain matures. When the domain is new or the issues will not necessarily recur, keeping a semi-formal case-based repository will suffice. This kind of process is in alignment with many development organizations that are using informal e-mail and discussion software, such as Lotus Notes, to disseminate and archive information. It is, in many ways, an extension of the traditional office memorandum and filing cabinet method. It is not until a domain has recurring problems or is part of the development network that more formal methods need to be applied. This allows a level of discrimination in terms of what kind of documentation and rigor is needed for a given problem. As opposed to applying rigorous methods universally, developers can place their time and effort where it is needed the most.

The other strength of the organizational learning approach is that it embraces diversity. Instead of trying to fit CASE tools into problems or domains they are not well-designed for, this approach seeks to learn which tools are best for given types of problems. This flexibility allows people to work with familiar tools and process models, but comes with the potential expense of having to train developers in a wide range of CASE tools. Our observations at UPRR indicate this happens anyway, but we are investigating ways in which the expertise of project members can provide an input to the process of choosing and using CASE tools.

3.2. From cases to domain knowledge

One problem with a case-based approach to software development is that cases tend to represent isolated problems that need to be adopted from one set of specifics to another. Domain analysis methods are needed that synthesize similar cases into knowledge that is applicable to a class of problems. This is precisely where organizational *learning* comes in. Most methods advocate creating a formal model of the domain (Prieto-Díaz, 1991), making domain analysis and software reuse most useful for established domains with well-known parameters (Biggerstaff, 1992). But in the fast-paced world of technological advances that characterizes the computer industry, well-established domains are an increasingly rare commodity. Domain analysis methods need to be able to accommodate the intrinsic forces of change stemming from the difficulty of creating well-designed systems to begin with, as well as meeting the needs of a dynamic marketplace that reflect changing and evolving user needs (Computer Science and Technology Board, 1990).

The real issue of domain analysis is to find commonalties among systems to facilitate reusing software and other design artifacts. From this perspective, domain analysis is a process of identifying commonly occurring patterns across a number of development efforts. The "domain" does not necessarily need to be a family of applications or a formal model, but a set of problems within applications with recurring activities and/or work products. As patterns emerge, top-down domain analysis methods can be used to formalize the patterns, facilitating domain evolution from the identification of isolated patterns to formally defined domain knowledge. Identifying established patterns of effort reduces the risk of costly domain analysis efforts by ensuring that the cost of analysis can be amortized over many uses.

We have been investigating techniques that help domain analysts detect recurring patterns with the case-based repository (Henninger et al., 1995). Spreading activation (Henninger, 1994) and analogy-based matching (Maiden and Sutcliffe, 1992) an be used to identify cases with potentially similar characteristics. For example, suppose an organization has a number of projects that have begun to struggle with issues of backup and recovery in a client-server architecture. The analyst begins by querying the system with terms such as "backup" and "recovery", finding characteristics such as "Automatic backup", "File backup", Disaster recovery", "Backup scheduling", "Update frequency" and others. The analyst consults the repository to understand some of the different ways backup and recovery have been addressed. From these, the analyst begins to construct facets (Prieto-Díaz, 1991) to help understand the domain, such as:

- mode: automatic or manual
- data type: database, files
- architecture: mainframe, server, workstation
- scheduling: volume size, loading

From here the analyst can begin to organize the software artifacts that have accumulated about backup and recovery issues. The repository provides a comprehensive and convenient mechanism for performing the analysis. It is precisely this kind to support for domain analysis that is necessary to provide "the reference assistance other types of engineers have benefited from for decades" (Computer Science and Technology Board, 1990)

3.3. Domain-specific design environments

Domain-specific design environments (Fischer and Lemke, 1988) integrate a domain-oriented framework with reusable components that can be selected and configured to automatically construct systems though the direct manipulation of visual icons representing code components. Systems are composed in a work area that is monitored by "critics" (Fischer et al., 1991a) that display artifact-centered, domain-specific, intelligent support when sub-optimal design decisions are detected (Fischer et al., 1992). These systems can provide a very high-level of specific support because they concentrate effort on good solutions within a domain, instead of addressing universal solutions. Fischer and his colleagues have also defined an incremental process of knowledge acquisition (Fischer et al., 1994). But this approach assumes that the domain knowledge is a known entity, ignoring important issues of identifying domains as they emerge in organizations and providing support when formal representations of design artifacts are not immediately available.

A key issue in the domain lifecycle is identifying the domains so that increasingly formal representations can be developed. For example, taking the domain analysis presented in the previous section, a design environment tool builder could begin the process of creating an environment for automatically creating a backup and recovery solution. Users would describe their application needs and choose elements of the facets identified in the domain analysis. The system would then critique the selections, guiding the developer toward quality solutions based on the accumulated domain experience of people in the organization. While the cost of creating of these kinds of domain-specific design environments may be relatively high, we are guaranteed that the environment will see use as its creation was based on an analysis of recurring problems in the organization.

4. Related work

In many respects the approach outlined here follows the domain analysis prescription to *identify* reusable information in the problem domain, *capture* relevant information, and *evolve* the information to meet current needs (Arango, 1989). Domain analysis techniques have been designed to systematically identify objects and relationships of a class of systems (Neighbors, 1984). But it would not be too unfair to characterize most domain analysis approaches as a form of top-down analysis that is difficult to apply unless the domain is well-understood to begin with (Arango, 1989; Biggerstaff, 1992). Our approach augments these efforts in two ways: (1) The entire domain lifecycle is supported, from its inception by trail-blazing projects to encoding domain abstractions in a design environment, not just the intermediate step of formalizing an already well-known domain. (2) The process of domain understanding is supported with tools that help identify recurring patterns of activities in an organization's software development efforts that can be flagged as candidates for formal domain analysis efforts.

The software factory approach (Caldiera and Basili, 1991) is similar to our domain lifecycle in that it separates developers into roles of application developers and reusable component developers. The STARS framework shares the concerns with developing and maintaining domain-specific assets for the continual improvement of reuse-oriented activities (STARS, 1992). The experience factory (Caldiera and Basili, 1991) is similar in spirit to our case-based repository of project experiences. Technology books formalize knowledge about algorithms for classes of problems (Arango et al., 1993). Case-based reasoning techniques have been employed to adapt and compose reusable components (Fouque and Matwin, 1993). While these methods largely focus on the component and algorithm levels, we take a broader view to include any significant development issue. For example, one project at UPRR performed a study of screen ergonomics. While the project was eventually canceled, the screen ergonomics report is highly regarded and has been used by other projects. Our objective is to provide a formalized process by which such artifacts can be identified and disseminated for widespread use.

Design rationale tries to capture the rationale behind the designs of systems (Lee, 1993). Tools like gIBIS (Conklin and Yakemovic, 1991), PHI (Fischer et al., 1991b), Sibyl (Lee, 1990), Remap (Ramesh and Dahr, 1994) and QOC (Maclean et al., 1991) use various representations to allow design alternatives to be collected and explored through browsing and retrieval methods. These methods advocate a process of deliberation in which design decisions are reached by consulting the design issues and alternatives stored in a repository. Our perspective is slightly different in that we are interested in capturing *what* occurred as a result of the decision, and *how* future efforts of a similar nature can be *improved*, not just capturing the rationale of *why* a system was designed a certain way.

Our approach is most closely related to some approaches to constructing organizational memory systems (Walsh and Ungson, 1991). While the organizational *learning* approach outline in this paper emphasizes the process of learning from and improving on previous efforts, organizational memory efforts focus on the first step in our domain lifecycle, collecting and disseminating design information. TeamInfo focused primarily on identifying categories of querying and browsing behavior for an organizational memory of loosely organized e-mail messages (Berlin et al., 1993). Answer Garden was built to turn knowledge into an organizational asset in a network of multiple-choice questions and answers (Ackerman and Malone, 1990). Their bottom-up process evolves the repository in response to user questions, and would be most useful for collecting experiences about development tools. Our framework goes further to formalize the process of analyzing domains and turning the individual cases into assets that can streamline the development process.

The Designer Assistant project at AT&T is similar in scope and philosophy to our approach (Terveen et al., 1995). They are not only interested in building support tools, but also designing *how the tool should be used.* Their Designer Assistant system was integrated with existing practices by fitting its use into the development process. This strategy not only ensured that Designer Assistant is used, but also that it is modified as part of the development process, allowing it to evolve with changing organization needs. The primary difference is the Designer Assistant only deals with the organizational memory aspects, leaving no support for domain analysis or the construction of domain-oriented design environments. Also, the Designer Assistant's repository is structured as an advice-giving system, while we have chosen to use case-based reasoning techniques that allow more open-ended queries that are needed for certain kinds of problem solving, and that have been requested in feedback from Designer Assistant users (Terveen et al., 1993).

5. Conclusions and future work

The technical innovations of the CASE field can reach their potential only through a shift that incorporates the broader organizational and methodological issues of software development. An approach in which experiences from previous and ongoing projects and brought to bear on the issues faced by development and maintenance projects is needed. We must begin to develop techniques to support the *domain lifecycle* (Simos, 1988), in which software development projects are seen as members of a product line or product family defined by an application domain. Routine development becomes routine, drawing on reusable designs, design processes, and other artifacts, leaving adequate resources to concentrate on the novel aspects of software system.

We support the domain lifecycle through CASE tools that define a progression from individual project experiences (cases) to domain-oriented design environments that formalize development knowledge and artifacts. Appropriate levels of formality is supported as an organization learns about the domains their software is built around. The development of novel activities is supported through project experiences accessed through case-based repository technology. As activities are repeated, case-based technology is employed to identify recurring development issues and support the process of generalizing from individual cases to domain-specific abstractions such as design guidelines, domain models and other formal structures. As knowledge of the domain accumulates, knowledge-based design environments can be created that automate design and provide intelligent support for design activities.

This approach is based on empirical observations of a large software development organization. We are currently in the process of integrating these tools into the organization, paying careful attention to existing development practices while creating the necessary infrastructure to take full advantage of the improvements offered by the domain lifecycle. We are in the process of empirically validating the domain lifecycle by working closely with an ongoing development project at UPRR. We have identified a domain that the project is working on that will have impact on other projects in the organizations. We are currently analyzing this domain by collecting cases from this and other projects working in this domain. The cases are being used to systematically analyze the domain and provide domain abstractions that can guide projects developing within this domain. We will also construct a design environment in the near future that will generate code on PC client platform used by the organization.

A key question that exists for our work as well as work in the areas of domain analysis, design rationale, and organizational memory, is how the process of generating and using repositories of design information can be embedded in the everyday practice of software development so that the repository evolves with the ever-changing goals and accomplishments of the organization (Terveen et al., 1995). Our approach advocates using the repository to both track project progress and as an information resource to support design and decision making. Much more work is needed to accomplish this goal. We are approaching this problem in a user-centered, participatory design, process in which we work with users (developers at UPRR) and deploy prototypes to collect feedback and refine our model to fit the organization's needs. Successful deployment of this system will not only help the software development process at UPRR, but will provide a crucial first step toward better understanding the software development process and how it can be improved.

Acknowledgments

I gratefully acknowledge the efforts of Lynden Tennison and Jim Montequin at Union Pacific Railroad. I am also indebted to a number of graduate students that have helped, including Kurt Baumgarten, Charles Daniel, Kyle Haynes, Kris Lappala, and Chris Neblett. This research was funded by the National Science Foundation (CCR-9502461), Union Pacific Railroad, and the Applied Information Management Institute in Omaha.

References

Ackerman, M.S. and Malone, T.W. 1990. Answer garden: A tool for growing organizational memory. In *Proceedings of the Conference on Office Information Systems*, pp. 31–39.

Arango, G. 1989. Domain analysis: From art form to engineering discipline. In *Proceedings Fifth International Workshop on Software Specification and Design*, Pittsburgh, PA, pp. 152–159.

Arango, G., Schoen, E., and Pettengill, R. 1993. A process for consolidating and reusing design knowledge. *15th International Conference on Software Engineering (ICSE'93)*, Baltimore, MD, pp. 231–242.

Belkin, N.J. and Croft, W.B. 1992. Information filtering and information retrieval: Two sides of the same coin?. *Communications of the ACM*, 35(12):29–38.

Berlin, L.M., Jeffries, R., O'day, V.L., Paepcke, A., and Wharton, C. 1993. Where did you put it? Issues in the design and use of a group memory. In *Proc. InterCHI'93*, Amsterdam, pp. 23–30.

Biggerstaff, T.J. 1992. An assessment and analysis of software reuse. *Advances in Computers*, 34:1–57.

Boehm, B.W. 1988. A spiral model of software development and enhancement. *Computer*, 21(5):61–72.

Caldiera, G. and Basili, V.R. 1991. Identifying and qualifying reusable software components. *Computer*, 24(2):61–70.

Card, D.N., McGarry, F.E., and Page, G.T. 1987. Evaluating software engineering technologies. *IEEE Transactions on Software Engineering*, 17(7):845–851.

Church, T. and Matthews, P. 1995. An evaluation of object-oriented CASE tools: The newbridge experience. *IEEE Seventh International Workshop on Computer-Aided Software Engineering—CASE'95*, pp. 4–9.

Computer Science and Technology Board. 1990. Scaling up: A research agenda for software engineering. *Communications of the ACM*, 33(3):281–293.

Conklin, E.J. and Yakemovic, K. 1991. A process-oriented approach to design rationale. *Human-Computer Interaction*, 6(3–4):357–391.

Curtis, B., Krasner, H., and Iscoe, N. 1988. A field study of the software design process for large systems, *Communications of the ACM*, 31(11):1268–1287.

Deerwester, S., Dumais, S.T., Furnas, G.W., Landauer, T.K., and Harshman, R. 1990. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407.

Domeshek, E.A. and Kolodner, J.L. 1992. A case-based design aid for architecture. *Artificial Intelligence in Design'92*, pp. 497–516.

Fischer, G. and Lemke, A.C. 1988. Construction kits and design environments: Steps toward human problem-domain communication. *Human-Computer Interaction*, 3(3):179–222.

Fischer, G., Lemke, A.C., Mastaglio, T., and Morch, A.I. 1991a. Critics: An emerging approach to knowledge-based human computer interaction. *International Journal of Man-Machine Studies*, 35(5):695–721.

Fischer, G., Lemke, A.C., McCall, R., and Morch, A. 1991b. Making argumentation serve design. *Human-Computer Interaction*, 6(3–4):393–419.

Fischer, G., Girgensohn, A., Nakakoji, K., and Redmiles, D. 1992. Supporting software designers with integrated, domain-oriented design environments. *IEEE Transactions on Software Engineering, Special Issue on Knowledge Representation and Reasoning in Software Engineering*, 18(6):511–522.

Fischer, G., McCall, R., Ostwald, J., Reeves, B., and Shipman, F. 1994. Seeding evolutionary growth and reseeding: Supporting the incremental development of design environments. In *Proc. of the Conference on Computer-Human Interaction (CHI'94)*, Boston, MA, pp. 292–298.

Fouque, G. and Matwin, S. 1993. A case-based approach to software reuse. *Journal of Intelligent Information Systems*, 1:165–197.

Gaines, B. 1989. Social and cognitive processes in knowledge acquisition. *Knowledge Acquisition*, 1(1):38–58.

Gomaa, H. and Kerschberg, L. 1995. Domain modeling for software reuse and evolution. *IEEE Seventh International Workshop on Computer-Aided Software Engineering—CASE'95*, pp. 162–171.

Granger, M. and Pick, R. 1991. Computer-aided software engineering impact on the software development process: An experiment. In *Proc. 24th Annual International Conference on System Sciences*, Hawaii, pp. 28–35.

Griss, M.L. 1993. Software reuse: From library to factory. *IBM Systems Journal*, 32(4):548–565.

Henninger, S. 1994. Using iterative refinement to find reusable software. *IEEE Software*, 11(5).

Henninger, S. 1995. Information access tools for software reuse. *Journal of Systems and Software*, 30(3):231–247.

Henninger, S. and Lappala, K. 1994. Finding the Right for the Job, UNL-CSE-94-002, University of Nebraska-Lincoln, Department of Computer Science and Engineering, Lincoln, NE.

Henninger, S., Lappala, K., and Raghavendran, A. 1995. An organizational learning approach to domain analysis. *Seventeenth International Conference on Software Engineering*, Seattle, WA, pp. 95–104.

Holtzblatt, K. and Jones, S. 1993. Contextual inquiry: A participatory technique for system design. *Participatory Design: Principles and Practice*, A. Namioka and D. Schuler (Eds.), Erlbaum, Hillsdale, NJ.

Huff, C.C. 1992. Elements of a realistic CASE tool adoption budget. *Communications of the ACM*, 35(4): 45–54.

Humphrey, W.S. 1989. *Managing the Software Process*. Reading, MA: Addison Wesley.

Kemerer, C.F. 1992. How the learning curve affects CASE tool adoption. *IEEE Software*, 9(3):23–28.

Kolodner, J.L. 1991. Improving human decision making through case-based decision aiding. *AI Magazine*, 12(1): 52–68.

Kolodner, J.L. 1993. *Case-Based Reasoning*. San Mateo, CA: Morgan-Kaufman.

Lee, J. 1990. SIBYL: A tool for managing group design rationale. In *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW'90)*, Los Angeles, CA, pp. 79–92.

Lee, J. 1993. Design rationale capture and use. *AI Magazine*, 14(2):24–26.

Maarek, Y.S., Berry, D.M., and Kaiser, G.E. 1991. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813.

Maclean, A., Bellotti, V., Young, R., and Moran, T. 1991. Questions, options, and criteria: Elements of design space analysis. *Human-Computer Interaction*, 6(3–4):201–251.

Maiden, N.A. and Sutcliffe, A.G. 1992. Exploiting reusable specifications through analogy. *Communications of the ACM*, 35(4):55–64.

Neighbors, J. 1984. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10:564–573.

Norman, R. and Nunamaker, J. 1989. CASE Productivity perceptions of software engineering professionals. *Communications of the ACM*, 32(9):1102–1108.

Poltrock, S.E. and Grudin, J. 1994. Organizational obstacles to interface design and development: Two participant observer studies. *ACM Transactions on Computer-Human Interaction*, 1(1):52–80.

Potts, C. 1993. Software-engineering research revisited. *IEEE Software*, 10(5):19–28.

Prieto-Díaz, R. 1991. Implementing faceted classification for software reuse. *Communications of the ACM*, 35(5).

Prieto-Díaz, R. and Arango, G. 1991. *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, Los Alamos, CA.

Ramanathan, J. and Sarkar, S. 1988. Providing customized assistance for software lifecycle approaches. *IEEE Transactions on Software Engineering*, 14(6):749–757.

Ramesh, B. and Dahr, V. 1994. Representing and maintaining process knowledge for large-scale systems development. *IEEE Expert*, 9(2):54–59.

Rich, C.H. and Waters, R.C. 1988. Automatic programming: Myths and prospects. *Computer*, 21(8):40–51.

Rieman, J. 1993. The diary study: A workplace-oriented research tool to guide laboratory efforts. *INTERCHI'93 Conference Proceedings*, Amsterdam, pp. 321–326.

Simos, M.A. 1988. The domain-oriented software life cycle: Towards an extended process model for reusability. In *Software Reuse: Emerging Technology*, W. Tracz (Ed.), IEEE Computer Society, Los Alamos, CA, pp. 354–363.

STARS, 1992. *Informal Technical Report for Software Technology for Adaptable, Reliable Systems (STARS)*. Electronic Systems Division, Report #STARS-UC-05159/001/00, USAF.

Terveen, L.G., Selfridge, P.G., and Long, M.D. 1993. From 'Folklore' to living design memory. In *Proceedings InterCHI'93*. Amsterdam, pp. 15–22.

Terveen, L.G., Selfridge, P.G., and Long, M.D. 1995. Living design memory—Framework, implementation, lessons learned. *Human-Computer Interaction*, 10(1):1–37.

Urban, J.E. and Bobbie, P.O. 1994. Software productivity: Through undergraduate software engineering and CASE tools. In *The Impact of CASE Technology on Software Processes*, D.E. Cooke (Ed.), World Scientific Publishing, Singapore.

Vesssy, I., Jarvenpaa, S.L., and Tractinsky, N. 1992. Evaluation of vendor products: CASE tools as methodology companies. *Communications of the ACM*, 35(4):90–105.

Walsh, J.P. and Ungson, G.R. 1991. Organizational memory. *Academy of Management Review*, 16(1):57–91.

Yourdon, E. 1992. *Decline & Fall of the American Programmer*. Englewood Cliffs, NJ: Yourdon Press/Prentice Hall.

Automated Software Engineering 3, 261–284 (1996)
© 1996 Kluwer Academic Publishers. Manufactured in The Netherlands.

A CASE Tool for Software Architecture Design

KENG NG kn@doc.ic.ac.uk
JEFF KRAMER jk@doc.ic.ac.uk
JEFF MAGEE jnm@doc.ic.ac.uk
Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK

Abstract. This paper describes the Software Architect's Assistant, an automated visual tool for the design and construction of Regis distributed programs. Unlike conventional CASE tools and their supported methodologies, the Architect's Assistant supports a compositional approach to program development in which the software architecture plays a central role throughout the software life-cycle—from the early design stage through to system management and evolution.

In its implementation, we have addressed some of the limitations of existing CASE tools, particularly in the degree of automated support offered to the human developer. Conscious effort has been made to maximise usability and efficiency, primarily by enhancing the level of automation and flexibility together with careful design of the user interface. Our objective is to provide a tool which automates all mundane clerical tasks, enforces program correctness and consistency and, at the same time, accommodates the individual working styles of its users.

Although currently specific to the development of Regis programs, the Architect's Assistant embodies concepts and ideas which are applicable to CASE tools in general.

1. Introduction

Configuration programming (Kramer and Magee, 1985) is a style of program development which emphasises the architectural view of a system. The premise of this approach is that a separate, explicit structural (configuration) description is essential for all phases in the software development process for distributed systems, from system specification as a configuration of component specifications to evolution as changes to a system configuration. Descriptions of the constituent software components and their interconnection patterns provide a clear and concise level at which to specify and design systems, and can be used directly by construction tools to generate the system itself. In many cases—particularly embedded applications—it is the structure of the application itself which is used to dictate the structure of the resultant system.

Regis (Magee et al., 1994) and its predecessors Conic (Magee et al., 1989) and REX (Kramer et al., 1992) are examples of systems which support this style of component-based program construction.

The prominence of the software architecture in this paradigm also makes it particularly well-suited for representation in a graphical form. Indeed, users of Regis programs usually design their programs graphically using pencil and paper before translating the hand-drawn diagrams into program text. Although useful, these paper diagrams do not offer the possibility of automation. Clearly, what is required here is a graphical front-end to the current

Regis environment that can provide life-cycle support for software development, from initial program design to the visualisation and evolution of the running program. The basic architecture of a Regis program, which is maintained throughout this life-cycle, would be a natural candidate as the unifying theme of such an environment. This then is the basic motivation for our work on the Software Architect's Assistant.

Our survey of existing CASE tools has failed to find any that is suited to our needs. Generic tools such as the Designer's Notepad (Haddley and Sommerville, 1990) shares many of our concerns on design capture and expression, and provides flexible means for note-taking, documentation and design exploration. However being a method-free tool, it is weak in method-specific automated support. Similarly, MacCadd (Logica, 1989) offers a configurable graphical editor with a pre-defined set of symbols (e.g., rectangles, circles) and connector types (dotted line, bold lines, arrows) but due to its generality, can offer only limited consistency checks. Similar limitations exists in other generic CASE tools such as Software Through Pictures (Wasserman and Pircher, 1987). Environments such as STATEMATE (Harel et al., 1988) recognise the power of the structural view for system specification and modelling, but tend to be weak in their support for component-based distributed system construction. STILE (STructured Interconnection Language and Environment) (Stovsky and Weide, 1987) advocates and provides good support for graphical component-based design and construction, but does not provide particular support for distribution. Graphical tools for parallel programming such as HeNCE (Beguelin et al., 1994) for PVM (Sunderam, 1990) and CODE 2.0 (Newton and Browne, 1992) share our belief in the benefits of structural visualisation, but adopts the dataflow model of computation and operate at a lower level of granularity (i.e., at the subroutine instead of component level). Similarly, Schooner (Homer and Schlichting, 1994) makes use of the AVS visualisation system (AVS, 1992) for configuring scientific applications made up of independently developed components. ObjecTime (1993) is targeted at distributed real-time systems and embodies many of the same concepts as Darwin/Regis, including the separation of system structure from its behaviour. As in Darwin, the system architecture is specified in terms of hierarchically-structured components which communicate through message ports. The dynamics of a system is modelled using a variation of Harel's StateCharts (Harel, 1987).

A common limitation we found in existing CASE tools is their failure to exploit the full potential of automation (Martin, 1988). We feel that the main goal of software tools should be to relieve the human developer of mundane, tedious bookkeeping tasks which are better performed by the computer than by a human. This will allow the designer to concentrate on the task at hand—the creative aspects of program development—instead of getting weighed down by routine clerical details. Unless tools are able to offer significant added value in terms of productivity and reduced costs, the provision of an electronic version of what was done with paper and pencil may be outweighed by the overhead of the learning curve.

The need for incorporating domain knowledge into development tools have long been recognised (Barstow, 1984; Rich and Waters, 1988) but this has only been partially realised in the tools we see today, mainly in the form of syntax-sensitive electronic sketchpads. For graphical CASE tools, we should embed not only knowledge of the methodology and notations, but also bookkeeping knowledge such as consistency rules and diagram layout preferences so that the tools can play the role of intelligent assistants to the human developer. We see this as a major challenge in our provision of visual tool support for Regis.

This paper outlines current research on an architectural framework for the engineering of parallel and distributed systems, and its associated support tool. In the following sections, we present a brief introduction to the Regis environment for component-based distributed programming, followed by a detailed description of the *Software Architect's Assistant*, an interactive graphical environment for the design and development of Regis programs. A simple case study—an active badge system (Harter and Hopper, 1994)—is used to illustrate the use of the tool, together with a discussion of some of its more novel ideas and features from a tool implementation viewpoint.

2. An overview of Regis

A Regis program consists of a set of loosely coupled, context-independent software components which communicate to achieve an overall goal. Hierarchical composition and decomposition is supported to enable complex components to be constructed out of simpler sub-components. In addition, these components may reside on the same machine or be distributed across a network of workstations.

2.1. Composite vs. primitive components

There are two kinds of components in Regis: primitive and composite. Primitive components are the basic computational components at the bottom of the program hierarchy, and are implemented in the C++ object-oriented programming language. A composite component, on the other hand, is constructed out of primitive components and other composite components. The structuring tool for the description of its structure is provided by the declarative configuration language Darwin. In other words, Darwin defines a composite component in terms of its internal components and the bindings between those components. An example of the graphical representation of a composite component and its corresponding Darwin code can be found in figure 10.

2.2. Component interface

The interface of a Regis component is described in terms of the services it provides to other components and those it requires of other components. These communication objects provide the means with which components interact with one another. For instance, the *sensornet* component in figure 10 communicates with its environment via the communication objects *sensout* and *sensin*. In the graphical notation used, services provided are drawn as filled circles whereas those required are drawn as white circles.

Interactions between components are represented by bindings which connect the communication objects required by one component to those provided by others. These are simply drawn as straight lines linking the corresponding objects.

A more in depth description of the Darwin language can be found in (Magee et al., 1993).

3. The Software Architect's Assistant

The Software Architect's Assistant is a visual programming tool which supports the design and construction of Regis distributed programs. It provides a framework in which software design can be captured, viewed and modified easily and quickly. Intelligent assistance is provided throughout the design process, from the sketching of the design diagrams to the generation of compilable Darwin code. Central to the Assistant is the Darwin sketchpad, a diagram editor with built-in knowledge of the Darwin syntax. Design diagrams are automatically tidied up to minimise crossovers between lines. Validation of the design is also supported, allowing specific program instances to be generated from the definition of generic software architectures. Support for the run-time monitoring and management of programs, as demonstrated by our work on ConicDraw (Kramer et al., 1989), will be integrated into the environment, hence enabling running programs to be visualised in the same graphical notation as in the original design. Compositional behaviour analysis tools (Cheung and Kramer, 1993; 1994) are to be integrated in the future.

The current implementation of the Assistant runs on the Apple Macintosh, and on Sun Solaris or HP/UX workstations under the Macintosh Application Environment (MAE).

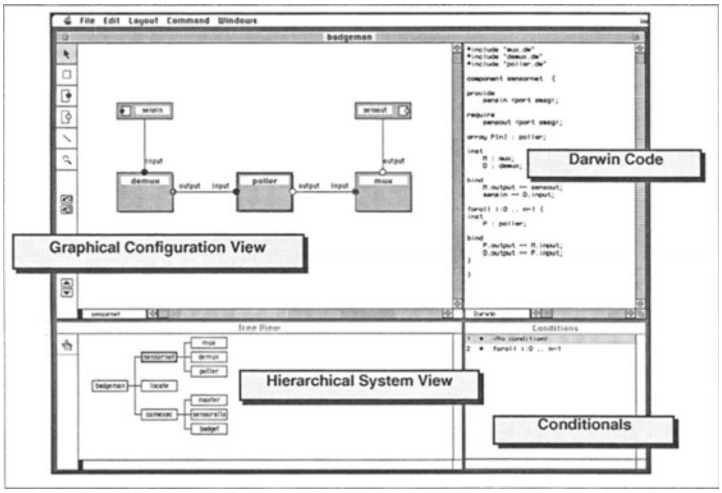
The rest of this paper highlights some of the novel features of the Assistant and the rationale behind some of our design and implementation decisions, particularly those which are pertinent to increasing usability and productivity. Using an active badge system as an example (Harter and Hopper, 1994), a step-by-step demonstration is then presented to illustrate its use in a typical scenario.

3.1. The user interface

The user interface of the Assistant is based on the theme of *structural visualisation*. This emphasis is especially relevant in the case of distributed programs where much of the underlying conceptual constructs are topological in nature, and can be naturally captured and presented in a graphical form (Harel, 1992). Furthermore, given that the basic software structure of many applications is fairly stable throughout the development process, it also provides an ideal framework for integrating the various software development and management activities and facilitates the presentation of a uniform and consistent user interface.

The Assistant allows the software architecture to be viewed from multiple integrated views (figure 1).

The Configuration Window contains a sketchpad in which the program architecture can be mapped out using the appropriate tools from the tool palette. The sketchpad displays the graphical configuration view of Darwin, and is where all editing of the program structure takes place. When drawing in the sketchpad, the designer is essentially defining a component *type* in terms of *instances* of components and the *bindings* between them. Controls are provided for traversing the hierarchy within this view. Multiple configuration windows can be opened to allow the viewing of different parts of the program at the same time. Each of these can in turn be split into 2 resizable viewing panes, with the right hand pane used for displaying, among other information, the Darwin code corresponding to the diagram being drawn.



```
#include "mux.dw"
#include "demux.dw"
#include "poller.dw"

component sensornet  {

provide
    sensin <port smsg>;

require
    sensout <port smsg>;

array P[n] : poller;

inst
    M : mux;
    D : demux;

bind
    M.output -- sensout;
    sensin -- D.input;

forall i:0 .. n-1 {
inst
    P : poller;

bind
    P.output -- M.input;
    D.output -- P.input;
}

}
```

```
1  •  <No condition>
2  •  forall i:0 .. n-1
```

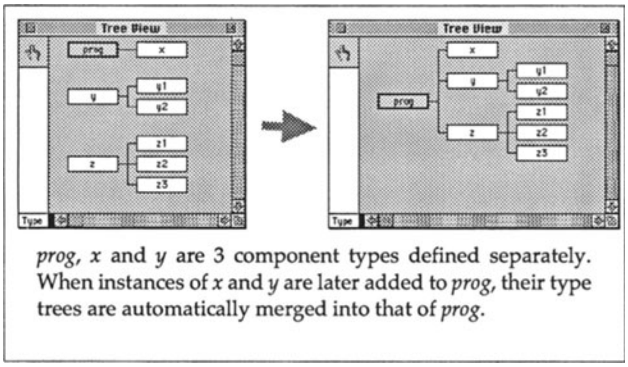
Figure 1. The main windows of the Architect's Assistant.

While each sketchpad allows the display and editing of a single component at a time, the associated Tree Window presents the entire program in a hierarchical tree structure. This is a type hierarchy which shows all the component types used in the program and the 'include' relationship between them. It is updated automatically whenever the program structure is modified in the sketchpad. It is useful not only as an indicator of the overall program structure but can also be used as a navigational aid for browsing the different parts of the system—selecting the appropriate type in this view will bring up the structure of that component type in the sketchpad. In addition, it serves as a 'where am I' indicator by highlighting the part of the tree diagram which corresponds to the component being displayed in the front-most sketchpad.

3.2. *Construction of program structure*

Structuring of the software architecture is one of the principal activities of software design. We feel strongly that no unnecessary restrictions should be placed on the software designer in order to accommodate the different working styles of users and also to encourage the exploration of alternative designs. Hence in the Assistant, both top-down and bottom-up development are supported as well as flexible ways of modifying the program hierarchy. This is in contrast with many CASE tools supporting hierarchical design notations which tend to impose a strict top-down design approach. We have also deliberately made the use of the Assistant flexible and informal by not enforcing strict adherence to any pre-defined workplan. For instance, there is no requirement to complete the specification of one component before starting on another.

In the Assistant, top-down decomposition is carried out by repeatedly elaborating the substructure of composite components. A component is expanded by double-clicking on



prog, *x* and *y* are 3 component types defined separately. When instances of *x* and *y* are later added to *prog*, their type trees are automatically merged into that of *prog*.

Figure 2. Merging type trees.

its box which zooms the user into its internal structure. Conversely, bottom-up construction is achieved by zooming up while at the root of the program hierarchy—a new root will be created at the top of the hierarchy as the parent of the current root. In practice, we find that a combination of top-down and bottom-up development is often needed in switching between the different levels of abstractions.

For a system like Regis, it is also not uncommon for a designer to begin work on the various component hierarchies in isolation and then bring them together to form the overall program. This is supported by allowing the creation of independent component hierarchies which can later be merged into the main program tree (figure 2).

If a component ever becomes too complicated, component instances within it can be easily grouped into a single component by dragging and dropping selected instances into a new instance. Any bindings to the selected instances will be automatically re-bound to the new one (figure 3). The reverse of this operation—the 'unwrapping' of a component to float its internal components to the level above—is also supported through a menu

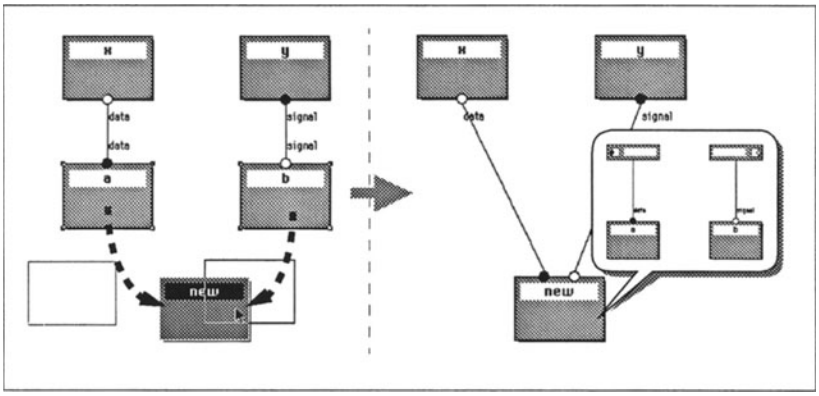


Figure 3. Grouping components.

command. Together, they allow the user to easily modify the groupings of components without incurring heavy penalties.

3.3. Program navigation

In supporting the viewing and editing of complex hierarchical structures, it is important that contextual information be provided so that it is clear where the current area of interest is in relation to the overall program. In the Assistant this function is served by the Tree view which automatically highlights the component being edited within the context of the program hierarchy. In addition, the shape and size of the tree give a good indication of the structure and complexity of the overall program.

The ability to move around the program hierarchy easily and quickly is also crucial. Within the sketchpad, navigation is controlled by the up/down buttons in the tool palette, or the up/down keys on the keyboard. Double-clicking on a component box also provides an intuitive alternative for dropping the user into the internal structure of the component. An alternative method for moving up the system hierarchy is to use the 'level' pop-up menu at the bottom left hand corner of the diagram window (see Appendix). Since the name of the current component is always visible in the menu, it also serves as an indication of where we are in the system hierarchy.

For navigation across different parts of the program hierarchy, the user should not be required to traverse through intermediate levels of the hierarchy (c.f., the traversal of hierarchical file structures such as that of Unix). For this task, the Tree view again fulfils an important role by allowing instant switching from one component to another through a simple mouse click. This is especially useful in the case of large programs with extensive hierarchies.

3.4. Support for diagram manipulation

Much effort has gone into making diagram editing within the sketchpad as painless a task as possible. This is important because diagram manipulation is often one of the most tedious aspects of program design with traditional CASE tools. Although most tools maintain the connectivity of diagram elements, few provide any further aid in the layout of diagrams. Our goal in this respect is to enable the desired program structure and layout to be achieved while requiring minimal effort from the user. We adopt a 'do what I mean' approach, allowing user actions to be imprecise and making it the tool's responsibility for carrying out what was intended in the first place.

To give diagrams a regular appearance, all component boxes drawn are made the same size by default. This also means that just a single mouse click is required for adding and placing a new component. Similarly, a provide or require port is added to a component by a single click anywhere within its box. The newly created port (in the form of a black or white circle) will automatically snap to the side of the box nearest to the mouse click, and all ports on that side will then be evenly redistributed along its length.

A binding is created by connecting a pair of provide/require ports from either direction. If a binding is drawn between a port and a component or vice versa, a complementary port will be automatically created for the component. Connecting a pair of component boxes

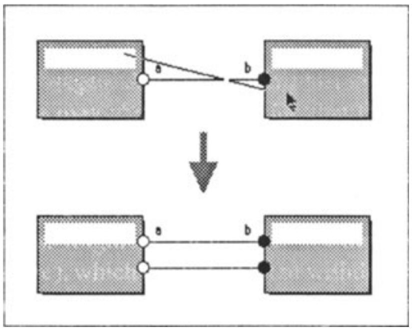


Figure 4. Creating a binding between 2 components.

will similarly result in the automatic creation of provide and require ports at the ends of the binding, and hence does away with the need to create ports separately before the bindings. Again, precision is not required as the user need only to draw a line anywhere within the 2 component boxes. The binding and its ports will be tidied up by the tool to avoid crossovers with existing bindings (figure 4). This automated layout facility is described in more detail in Section 3.5.

The Assistant provides dedicated tools in the tool palette for the creation and editing of the different types of Darwin objects. While this is an intuitive interface for tool selection and invocation, it can be inefficient and distracting due to the time wasted in moving the mouse between the drawing area and the palette for tool switching. We have tackled this problem in 2 ways. The first provides a quicker interface for tool selection which dispenses with the need to use the mouse: the 'tab' key on the keyboard advances tool selection in the palette, which 'wraps round' to the first tool when the last tool in the palette is reached. Hitting 'shift-tab' selects the previous tool. This use of the keyboard for tool selection enables a two-handed approach to diagram editing: the mouse can remain in the drawing area to edit the drawing while the other hand selects the appropriate tool.

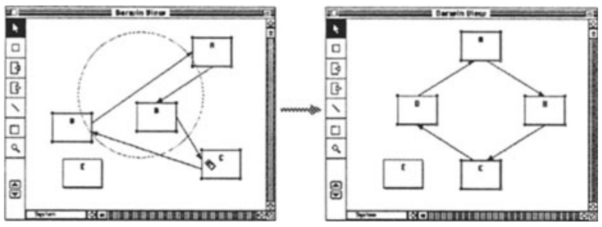
The second approach reduces the need for tool switching by overloading the functionality of the tools. For example, the provide tool can be used for creating a require port by holding down the option key, and vice versa. Furthermore, if clicked on the background of the sketchpad, the provide and require tools add the corresponding services (e.g., ● location) to the component type being edited, hence dispensing with the need for additional tools in the palette. In a similar vein, the arrow tool, normally used for moving and resizing, also doubles up as the 'magnet' tool for box alignment (see Section 3.5) as well as a short-cut for editing the information associated with Darwin objects (double-clicking on the name of an object brings up the corresponding Info dialog for that object).

3.5. Automated diagram layout

A comprehensive set of automated and manual aids is provided for the editing of Darwin configuration diagrams. Two levels of automated support are available. The first deals with the biggest problem of graph layout—the untangling of crossovers between bindings. The Assistant does this by relocating the provide and require ports at the ends of bindings and then distributing them along each face of a component box to give a regular appearance. The component boxes themselves are left unchanged. The algorithm used is based on our earlier work on ConicDraw but with its overall speed greatly improved by only cleaning up the parts of the diagram which have been modified since the last tidy-up. This makes it feasible for the operation to be invoked after every modification to the diagram layout such as the moving or resizing of a box. Consequently, configuration diagrams in the sketchpad are always in the tidied state. This facility has proven to be a major time-saver as it means that the only manual layout activity to be performed by the user is the placement of the component boxes, which determines the overall structure of a diagram.

The second level of automated support takes care of the auto-placement of the component boxes in a configuration graph, with the objective of producing a compact diagram with minimal crossovers between bindings and component boxes. The heuristic-based algorithm is used to generate the initial diagrams of imported Darwin code, and is also useful in situations where the large number of components and bindings makes it difficult for the human eye to easily come up with an aesthetically pleasing layout. The drawback, however, is that it is liable to produce 'unstable' layout, i.e., a small modification to the diagram may cause the algorithm to produce a drastically different layout. As a result, this facility is invoked only on demand by the user.

A common operation in diagram layout is the alignment of the component boxes. To simplify this task, the sketchpad has a built-in invisible alignment grid which constrains the placement of nodes in a diagram. When a component is created or subsequently moved, its top-left corner automatically snaps to the nearest point in the grid. Hence great precision is not required when placing and aligning components. The coarseness of the grid can be set by the user or the grid can be turned off altogether to permit finer control over the layout of a diagram. The 'magnet' tool (figure 5) is a further aid for aligning groups of components in either rows, columns or rings.



To place the 4 boxes A, B, C and D into a ring, the user first selects them and then draws a circle with the 'magnet' tool. The boxes will be pulled onto the circle and distributed evenly along its circumference. As with all other layout changes, the bindings in the diagram are automatically cleaned up after the operation.

Figure 5. Aligning components with the magnet tool.

3.6. Consistency management

Much work in software design are bookkeeping tasks, ensuring that information defined in one part of the system is consistent with that in another. These are tasks which are accomplished much quicker by the tool and with fewer errors. With the Assistant, we have attempted to automate all such bookkeeping work to ensure that consistency is maintained at all times. Whenever possible, any required information is inferred automatically from existing data, thus obviating the need for data re-entry and any subsequent error checking that entails. For example, if a component's instance is altered in one part of the program, the change is immediately reflected in all other instances of the same type.

Apart from structural consistency, the Assistant also maintains the consistency of data types of connected ports. For instance, changing the data type of a port will cause the new information to be propagated to all ports which are bound to it throughout the program hierarchy. When two ports are connected and one is undefined, the data type of the defined port will be copied automatically to the undefined one as well as all other ones to which it is connected. Connecting 2 incompatible ports results in a warning and the option of either copying the port type of the provide port to the require port, or vice versa. This time-saving facility is particularly useful when the program hierarchy is large, where the user would otherwise have to go through the entire hierarchy to make the changes manually.

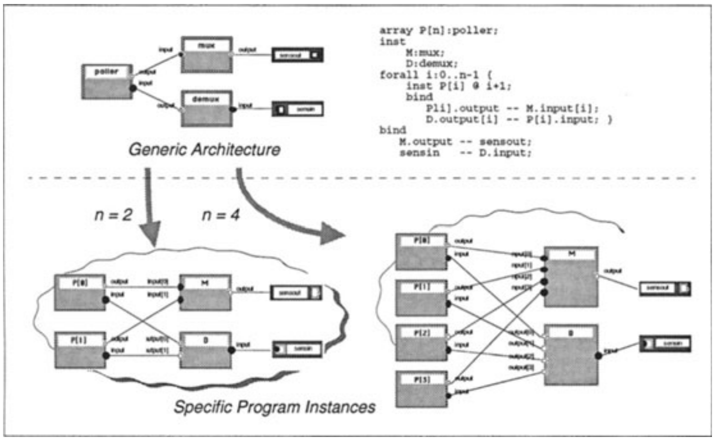
Consistency within the Assistant is enforced at the point of data entry, hence there is no need for a separate error checking phase—all accepted information is consistent at any point in time.

3.7. Design validation

A Darwin specification essentially describes a generic structure of a component type. The instantiated structure at run-time is determined by the actual parameters passed to the component instance and the evaluation of its conditional guards, and can be difficult to visualise at design time. As a design validation aid, the Assistant facilitates the testing of component descriptions against different parameter values, presenting as results the instantiated configurations of the component which are reflected back to the user in a graphical form (figure 6). This kind of 'what-if' scenario testing should allow many errors to be caught at an early stage without needing to go through the full compilation and execution cycles of program development. It is particularly useful for the validation of complex components which make use of Darwin's more advance facilities such as the parameterisation of components, replication of component instances and communication objects, conditional configurations with guards, and even recursive definition of components.

3.8. Data entry and retrieval

Much information is generated during the system development process. Apart from the 'core' information—that is the actual specification, design and code of the system—there is a huge amount of related information that needs to be maintained. Project status and history, design decisions, non-functional requirements, comments for source code, version control information are typical examples.



```
array P[n]:poller;
inst
    M:mux;
    D:demux;
forall i:0..n-1 {
    inst P[i] @ i+1;
    bind
       Pli].output -- M.input[i];
       D.output[i] -- P[i].input; }
bind
   M.output -- sensout;
   sensin   -- D.input;
```

Figure 6. Generating program instances from a generic software architecture.

A clear and uniform approach to information capture and retrieval is vital if the user is to be able to enter new information or retrieve existing data easily without needing to worry about unnecessary details such as where and how the information is stored in the file system. Equally important would be a navigation system which can take the user through the complex information space consisting of varied but related pieces of data. Conceptually, within the Assistant, we use the Darwin configuration diagrams as the basic organisational structure onto which information, or *attributes*, can be attached (figure 7). This allows all information related to an object to be 'packaged' with the object's representation on screen. It also means that the same navigational aids for the traversal of the program hierarchy is used for traversing the attribute information space. An object attribute is retrieved using the attribute tool through a pop-up menu (figure 7), and is displayed on the right-hand pane of the sketchpad window. A component's Darwin description is treated as an attribute that belongs to that component.

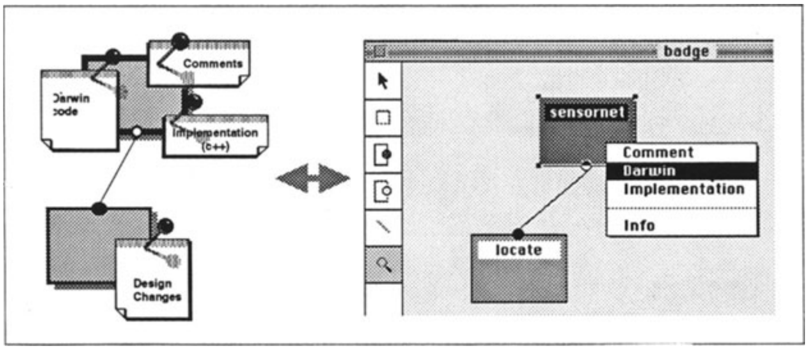


Figure 7. Component attributes.

The Assistant's attribute mechanism is extensible to allow capture of attribute types not originally envisaged. Entry of attribute details is optional so the user can enter as much or as little information for any particular project. Although currently limited to textual attributes, we hope to provide support for other kinds of structured data such as graphics, forms and charts. Attributes entered by the user are automatically collated by the Assistant and written out as a formatted report. The report includes, for each composite component type, its configuration diagram and Darwin code together with all attributes belonging to its sub-components. For a primitive component, the report includes its implementation in C++. All diagrams are automatically scaled to fit the printed page if necessary.

By automating information collation and formatting, the Assistant's report generator takes much of the drudgery out of report preparation. Since the report is written in the Rich Text Format (RTF) which is readable by most popular commercial word processors, further editing to the contents and layout is possible after the report has been generated.

4. Case study—An active badge system

An active badge system has been implemented in the Regis programming environment. Active Badges emit and receive infrared signals which are received/transmitted by a network of infrared sensors connected to workstations. The system permits the location and paging of badge wearers within a building.

In our implementation, the top-level of the badge system is made up of 3 Darwin components, namely *comexec*, *sensornet* and *locate* (figure 9). *Location*, *where*, *trace* and *command* are the open systems interfaces of the badge system which enable communication with external programs, and are to be registered with a name server (Kramer et al., 1989). Bindings may be made dynamically by a third-party (configuration manager) at runtime.

Comexec provides the badge command execution service—commands are issued to badges to set off its internal beeper or to illuminate LEDs. The Darwin component interface specification requires the specification of the types of these services (enclosed in angle brackets). By convention, the first word of the type specification is the interaction mechanism class. For example, command accepts entry calls with a request of type *comT* and a reply of *repT*.

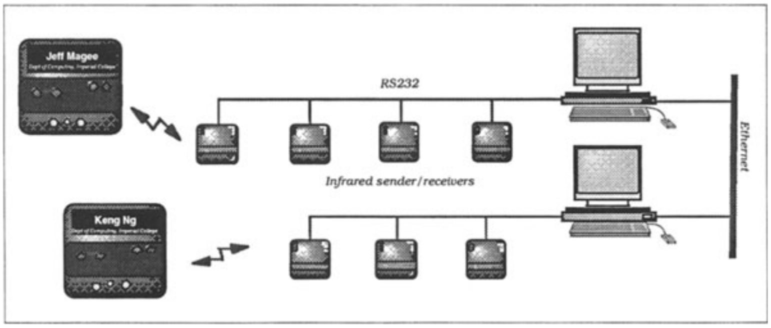


Figure 8. The active badge system.

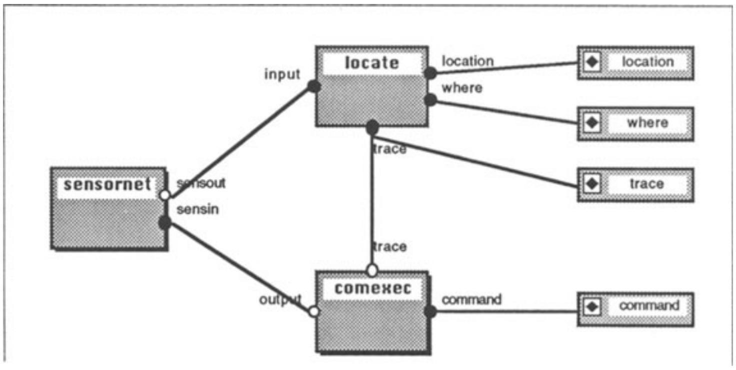
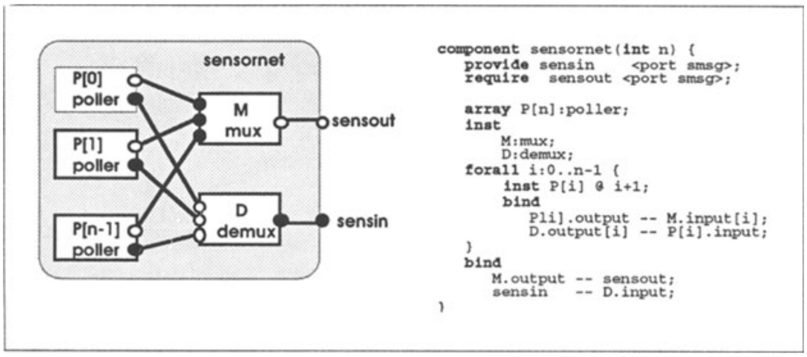


Figure 9. The Regis implementation of the badge system.



```
component sensornet(int n) {
   provide sensin    <port smsg>;
   require  sensout <port smsg>;

   array P[n]:poller;
   inst
       M:mux;
       D:demux;
   forall i:0..n-1 {
       inst P[i] @ i+1;
       bind
          Pli].output -- M.input[i];
          D.output[i] -- P[i].input;
   }
   bind
      M.output -- sensout;
      sensin   -- D.input;
}
```

Figure 10. The sensornet component.

To execute a command, it is necessary to first locate a badge. Consequently, *comexec* requires the location service which is provided by the component *locate*. Location information in the badge system is an event stream where an event represents a change of badge location. Thus the interaction mechanism for location is *event* and the data type of each event is *bstatus*. Similarly, to execute the badge once found, the component must send a message to the sensor network. The requirement for this service is represented by *output* which uses the Regis port message transmission primitives. Note that the component *comexec* does not need to know the names of external services or where they may be found. It may be implemented and tested independently of the rest of the badge system. We call this property context independence. It permits the reuse of components during construction and simplifies replacement during maintenance.

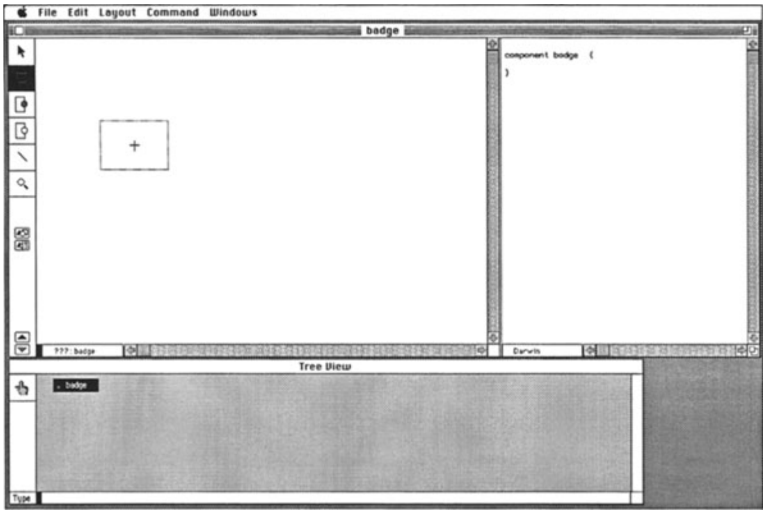
The composite component *sensornet* controls the interface to the network of infrared badge sensors. Each requirement (empty circle) in this case is for a port (named *output*) to send messages to, and each provision (filled in circle) is a port from which a component receives messages (named *input*). Requirements which cannot be satisfied inside the component can be made visible at a higher level by binding them to an interface requirement as has been done in the example for multiplexer M requirement output which is bound to *sensout*. Similarly services provided internally which are required outside are bound to an interface service provision e.g., *sensin—D.input*.

Each *poller* component is located on a different workstation and controls a multidrop RS232 line of sensors. The *poller* component requires a service to output badge location sighting messages and provides an input on which to transmit command messages. In general, many requirements may be bound to a single provided service. However, in this case each poller instance output is bound to a separate input port to allow the multiplex component M to identify the sensor network in the outgoing message. Pollers are distributed by the expression $instP[i]@i + 1$ which locates each instance $P[i]$ on a separate machine $i + 1$. The integer machine identifiers are mapped to real workstations by the Regis runtime system. The mapping permits program portability.

From the example, it can be seen that components may be parameterised and that parameters can be used to determine the internal structure of composite components. In this case the parameter determines the number of poller instances.

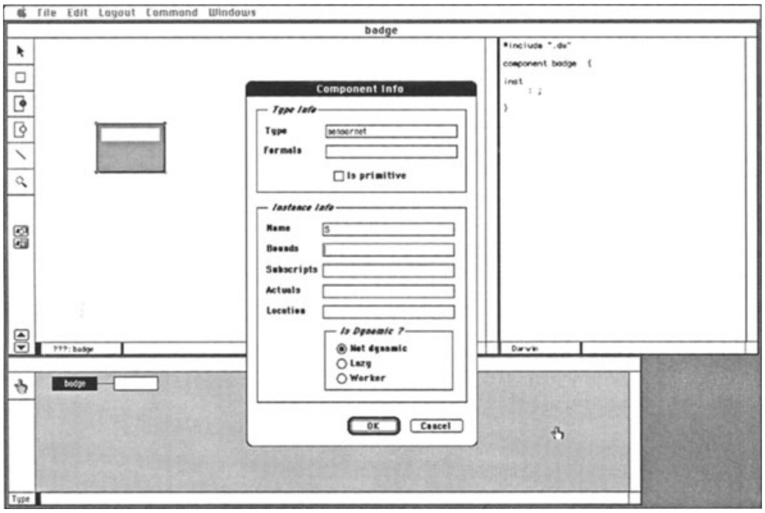
5. Constructing the badge system with the Architect's Assistant

In this section we present a step-by-step demonstration of how the Architect's Assistant is used in the construction of the active badge system described above.



```
component badge  (
)
```

Creating a component. Initially, the Assistant presents a blank or undefined type which corresponds to the 'root' of the program being built. We have named the program 'badge', as indicated by the name of the sketchpad window. The badge component type is elaborated by sketching out its internal structure within this window. A component instance is created by drawing a box with the *component* tool. In this case we are creating an instance of the *sensornet*.



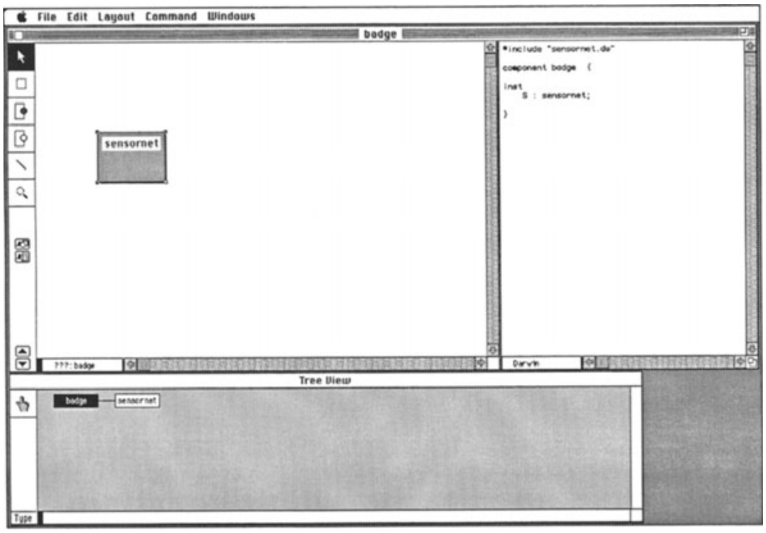
Type Info
Type: sensornet
Formals:
[ ] Is primitive

Instance Info
Name: S
Bounds:
Subscripts:
Actuals:
Location:

Is Dynamic ?
[✓ Not dynamic]
Lazy
Worker

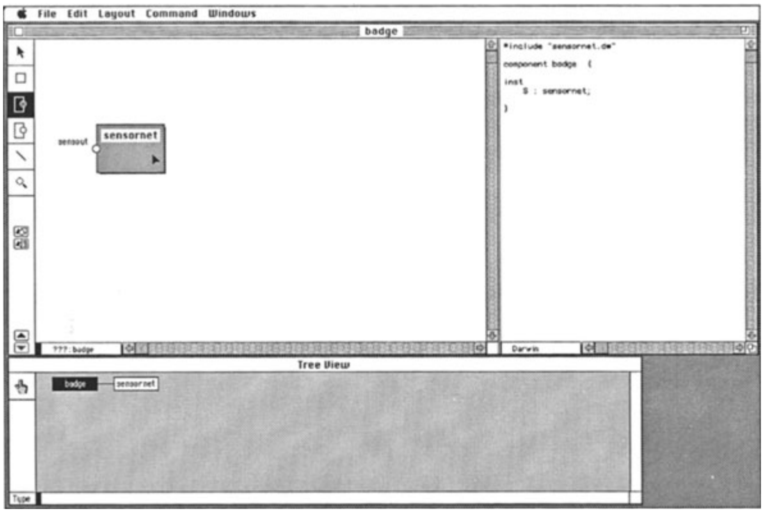
```
#include ".dw"
component badge {
inst
    : ;
}
```

Naming a component. Information related to this component, such as its type and instance name, array bounds, actual and formal parameters, is entered through a dialog box. Any of this information can be left undefined initially and filled in later.



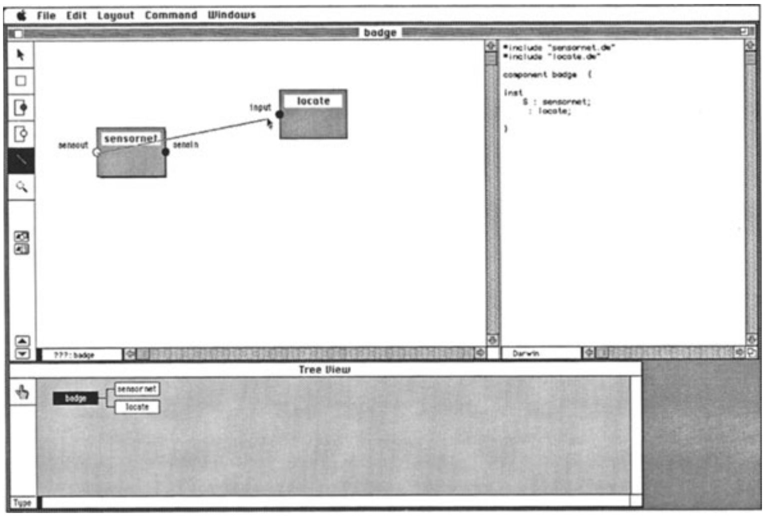
```
#include "sensornet.dw"
component badge {
inst
    S : sensornet;
}
```

Naming a component (2). After despatching the dialog, the component's type name will be displayed within its box as well as in the tree view. Options are available for displaying the component instance name and/or its type name within the sketchpad. Note that the Darwin code to the right of the sketchpad has been updated automatically.



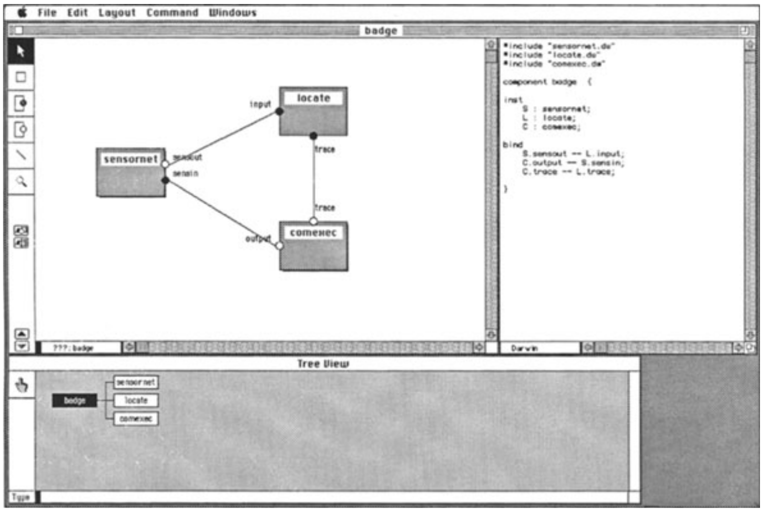
```
*include "sensornet.dw"

component badge  {

inst
    S : sensornet;

}
```

Editing component interface. Communication objects for *sensornet* can be created explicitly with the *provide* or *require* tools by clicking anywhere inside its box. It will be fixed to the nearest side of the box.



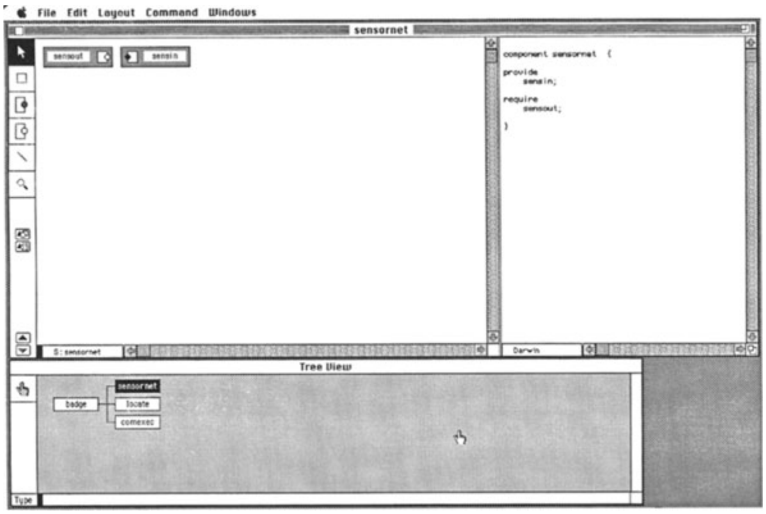
```
*include "sensornet.dw"
*include "locate.dw"

component badge  {

inst
    S : sensornet;
     : locate;

}
```

Creating a binding. Bindings are created by drawing a line between a pair of complementary provide/require objects. These snap to the appropriate sides of the components and are automatically positioned to avoid any crossovers with existing bindings. Alternatively, if we connect a pair of components, communication objects are automatically generated on the components at each end of the binding.



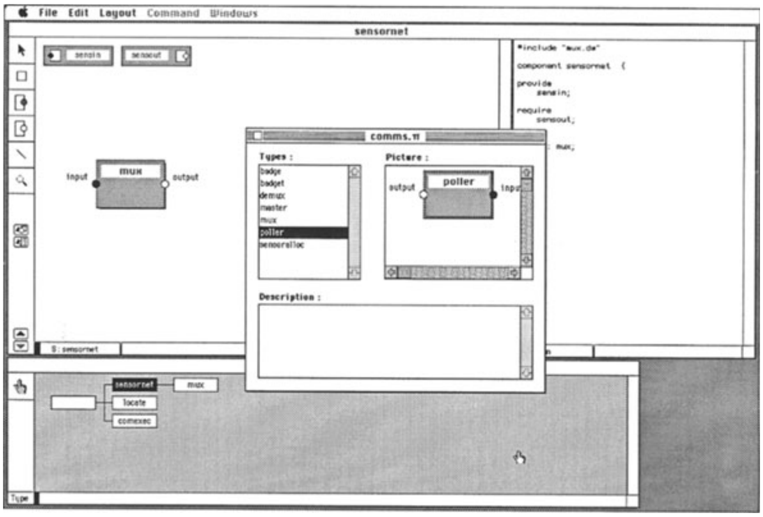
```
#include "sensornet.dw"
#include "locate.dw"
#include "comexec.dw"

component badge  {

inst
     S : sensornet;
     L : locate;
     C : comexec;

bind
     S.sensout -- L.input;
     C.output -- S.sensin;
     C.trace -- L.trace;

}
```

Refining a component. This shows the state of the badge component after *comexec* and more bindings have been added. Composite components are drawn as shadowed boxes, as opposed to plain boxes for primitive components.



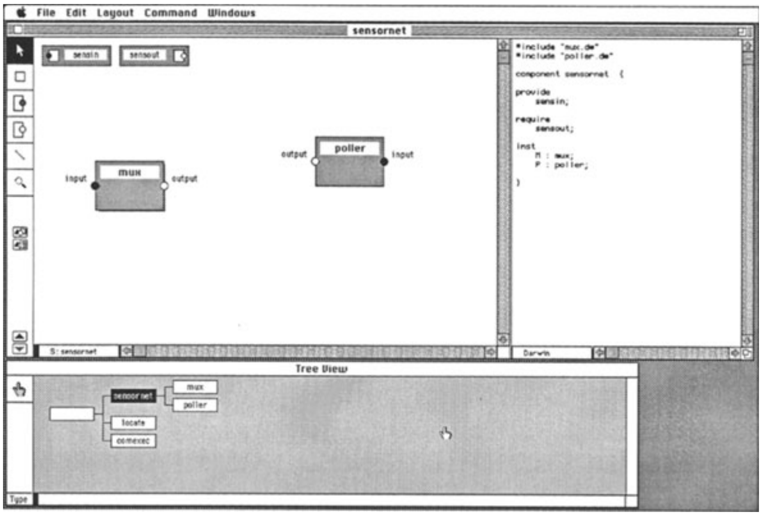
```
component sensornet  {

provide
     sensin;

require
     sensout;

}
```

Refining a component (2). To create sub-components within *sensornet*, we first double-click on its box. This drops us inside the component which is initially empty, apart from its interface which is represented by the interface boxes *sensin* and *sensout*. These are generated automatically by the Assistant based on what has previously been defined at the level above. The structure of *sensornet* can now be elaborated as descibed earlier.



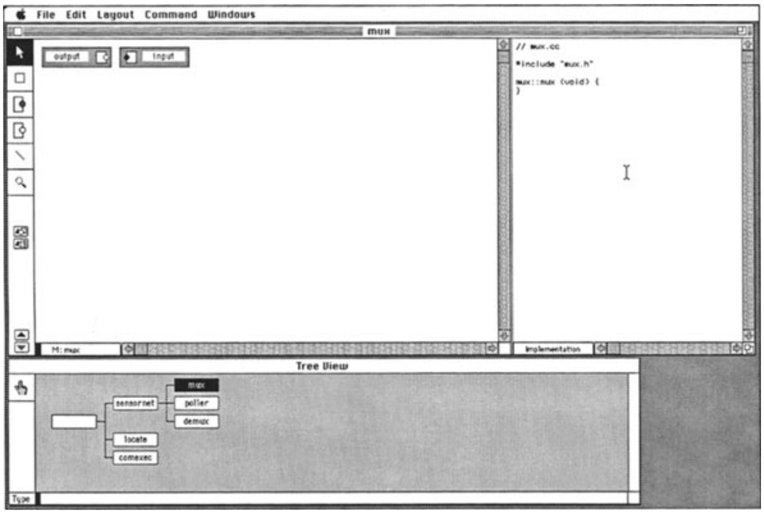
```
*include "mux.de"

component sensornet  {

provide
     sensin;

require
     sensout;

            : mux;
```

Reusing a component. The *poller* component needed for *sensornet* has previously been defined in another program (comms.π). It can be reused using the library browser of the Architect's Assistant. This presents a list of all component types defined in *comms.π* together with their corresponding diagrams and descriptions.



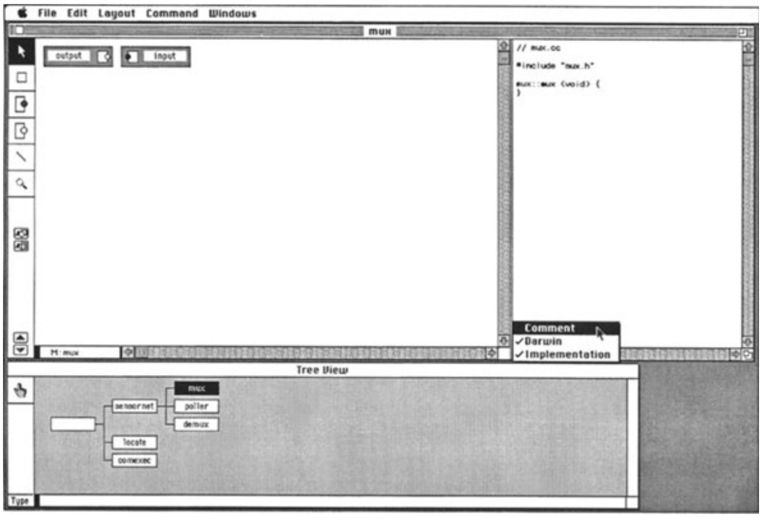
```
*include "mux.de"
*include "poller.de"

component sensornet  {

provide
     sensin;

require
     sensout;

inst
     M : mux;
     P : poller;

}
```

Reusing a component (2). Using the standard copy-and-paste technique, *poller* can be copied from the library and included in the *sensornet* component under development. By reusing a component, we pick up not only its diagram and code but also any additional information associated with it.



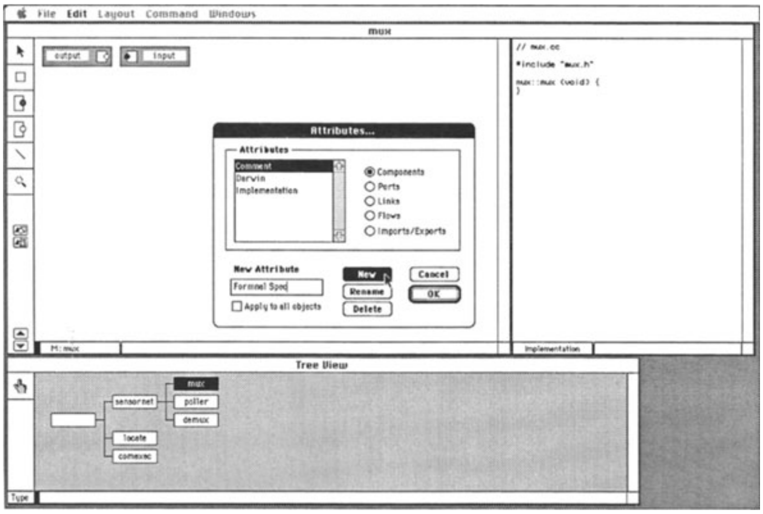
```
// mux.cc
#include "mux.h"
mux::mux (void) {
}
```

Defining a primitive component. The implementation of a primitive component (in the form of C++ code) is displayed to the right of a sketchpad when we enter such a component. Skeleton C++ code in the form of its class constructor is generated automatically by the Assistant for further elaboration by the user. Here we have entered the primitive component *mux*.



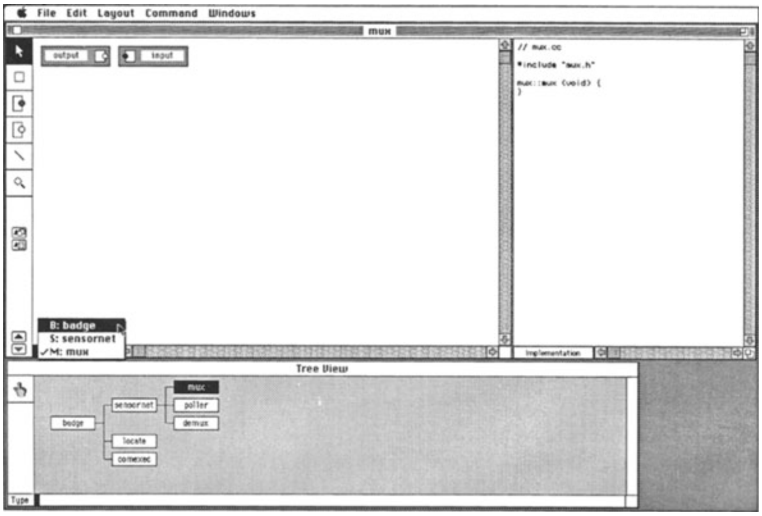
```
// mux.cc
#include "mux.h"
mux::mux (void) {
}
```

Attaching attributes to components. The right-hand pane of the sketchpad window displays attributes associated with the corresponding component in the sketchpad. *Darwin code*, *c++ code* (i.e., implementation) and *comments* are built-in attributes of components. The attribute pop-up menu allows switching between the different attributes.



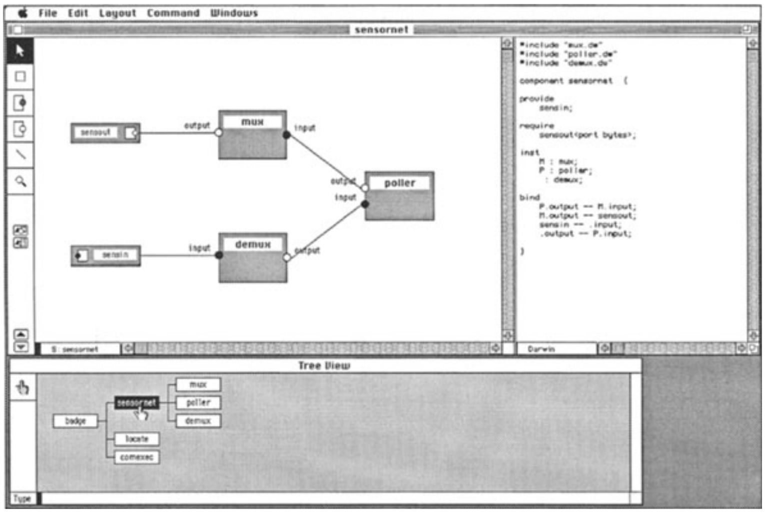
```
// mux.cc

#include "mux.h"

mux::mux (void) {
}
```

Creating new attribute types. New attribute types can be created to allow entry of new kinds of information. To facilitate the recording of formal specifications with Darwin components, we enter the name of the attribute type ('Formal Spec') in the attribute dialog. From then on 'Formal Spec' will appear as a new item in the attribute pop-up menu, thus allowing the display and entry of formal specifications within the attribute pane.



```
// mux.cc

#include "mux.h"

mux::mux (void) {
}
```

Navigation. There are several ways of traversing the program hierarchy, one of which is via the pop-up menu at the bootom-left of the sketchpad. This shows the path from the root of the hierarchy to the current component. Selecting an item from this menu switches the display of the sketchpad to this component.



```
#include "mux.dw"
#include "poller.dw"
#include "demux.dw"

component sensornet  {

provide
     sensin;

require
     sensout<port bytes>;

inst
     M : mux;
     P : poller;
       : demux;

bind
     P.output -- M.input;
     M.output -- sensout;
     sensin -- .input;
     .output -- P.input;

}
```

Navigation (2). Alternatively, the tree view also serves as a navigation tool. Clicking on a component, in this case *sensornet*, will switch the display of the sketchpad directly to the corresponding component.

6. Conclusion

In this paper we have described the Software Architect's Assistant, a graphical CASE tool for the design and construction of distributed systems. The Assistant encourages a constructive approach to program design (Kramer et al., 1990) in which systems are built up through the composition of software components. Unlike other methodologies in which the design architecture is discarded after system construction, the software architecture of a Regis system is maintained from the early design stage right through to the evolution of the running program. This emphasis on structure is mirrored in the Assistant which employs the visualisation of software architecture as the centrepiece of its user interface. It provides an ideal structuring tool for organising development information, a skeleton onto which information can be attached. At the same time, it allows the underlying data storage mechanism to be hidden from the user.

Within the ARES[1] project, we are currently investigating the use of the Darwin language to provide support for variance in the architecture of product families. An automated tool such as the Assistant will allow the instantiated architectures of the family members to be visualised and validated before the eventual systems are constructed.

From a tool implementation's perspective, our main concern has been to maximise usability and efficiency. Previous experience as both builders and users of software tools have convinced us that the key to this goal is to extend the level and scope of the following:

Automation. In many ways, software developers have failed to exploit the full potential offered by the increasing power of computer hardware. Our approach to tool building has been to offload from the user all tasks that can be performed by the machine. CASE tools should play the role of intelligent assistants, working with the human designer and automating any bookkeeping task that does not require human intervention. Only then can the full benefits of tool deployment be achieved, in terms of productivity and quality improvement and reduction in human effort and errors. The automated facilities for diagram layout, consistency management and report generation are all part of our effort towards this goal.

Flexibility. Although method-specific tools should provide guidance to help the user towards a design solution, it must not do so in a way that constrains his style of working. The user should be free to use his own initiative and follow any sensible design paths. Strict conformance to any prescribed workplan should be avoided in order to encourage exploration of alternative design and accommodate individual designer's working styles. Experience has shown that tools which put the user in a straight-jacket will not gain widescale use. In the design of the Assistant, a conscious decision was made to relax the rules governing when and in what order operations can be performed. The user is free to work in a top-down or bottom-up manner, or a combination of both. The entry of partial, incomplete information is also permitted, allowing different parts of the system to be refined and elaborated concurrently. Where it is sensible, we have also allowed the same information to be entered in different locations. For instance, the interface of a component can be defined either when defining the internal structure of the component type itself, or on an instance of that component type.

Speed. The performance of a software tool is critical to its success. This is especially true for visual, interactive tools where a delay of a fraction of a second will often be perceptible. A sluggish tool not only aggravates user frustration but tends to break up the user's flow of thought and concentration. In building the Assistant, we have endeavoured to optimise its performance in all areas, from the design of efficient data structures and algorithms to the exploitation of machine idle time and scheduling of background tasks so that any potential delays are kept to a minimum and hidden from the user. The automatic tidy-up of bindings, for example, is performed at idle time so that it does not interfere with user operations. Similarly, Darwin code generation is done only when the code view is visible and in idle time. In practice, however, these operations are usually completed so quickly they appear to the user as though they are performed immediately after every diagram modification.

The importance of a well-designed user interface cannot be underplayed. For a tool which spans several phases of the development process, a uniform and consistent interface helps to ease the learning curve and maintain the user's orientation across the different stages. It will also help to strengthen the sense of integration between the different software development and management tasks. The intended mental model can be further reinforced through a user interface modelled closely on the subject matter, along with carefully designed data representation and the shielding of unnecessary details.

Although the Architect's Assistant has been specially tailored for Regis distributed programs, most of the ideas and concepts it embodies are equally applicable to a wide class of CASE tools, especially those supporting design methods with structured, hierarchical graphical notations.

Note

1. ARES (Architectural Reasoning for Embedded Software) is an ESPRIT project funded by the Commission of the European Communities (CEC).

Acknowledgments

We gratefully acknowledge the advice of our colleagues in the Distributed Software Engineering Research Section, and the financial support provided by the CEC (ESPRIT project ARES-20477), EPSRC (grant ref. GR/J 52693 and GR/J 87022) and the DTI (grant ref. IED4/410/36/002).

References

AVS, 1992. AVS Developer's Guide (Release 4.0). Advanced Visual Systems Inc. Part n. 320-0013-02, Rev B.

Barstow, D.R. 1984. A perspective on automatic programming. *AI Magazine*, 5(1):5–27.

Beguelin, A., Dongarra, J., Geist, G., Manchek, R., and Sunderam, V. 1994. HeNCE: A users' guide (Version 2.0). Carnegie Mellon University.

Cheung, S.C. and Kramer, J. 1993. Enhancing compositional analysis with context constraints. Presented at *Symposium on the Foundations of Software Engineering*, Los Angeles, pp. 115–125.

Cheung, S.C. and Kramer, J. 1994. Tractable dataflow analysis for distributed systems. *IEEE Transactions on Software Engineering*, 20(8):579–593.

Haddley, N. and Sommerville, I. 1990. Integrated support for system design. *Software Engineering Journal*, pp. 331–338.

Harel, D. 1987. A visual formalism for complex systems. *Science of Computer Programming*, Vol. 8.

Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., and Shtul-Trauring, A. 1988. STATEMATE: A working environment for the development of complex reactive systems. Presented at *10th International Conference on Software Engineering*, pp. 396–406.

Harel, D. 1992. Biting the silver bullet—Toward a brighter future for system development. *IEEE Computer*, 25(1):8–20.

Harter, A. and Hopper, A. 1994. A distributed location system for the active office. *IEEE Network Special Issue on Distributed Systems for Telecommunications*.

Homer, P., and Schlichting, R. 1994. Configuring scientific applications in a heterogeneous distributed system, Presented at *2nd International Workshop on Configurable Distributed Systems*, Pittsburgh, pp. 159–168.

Kramer, J. and Magee, J. 1985. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, SE-11(4):424–436.

Kramer, J., Magee, J., and Ng, K. 1989. Graphical configuration programming. *IEEE Computer*, 22(10):53–65.

Kramer, J., Magee, J., and Finkelstein, A. 1990. A constructive design approach to the design of distributed systems. Presented at *10th International Conference on Distributed Computing Systems*, Paris, pp. 580–587.

Kramer, J., Magee, J., and Sloman, M. 1992. Configuring distributed systems. Presented at *5th ACM SIGOPS Workshop on Models and Paradigms for Distributed Systems Structuring*, Mont Saint-Michel, France.

Kramer, J., Magee, J., Sloman, M., and Dulay, N. 1992. Configuring object-based distributed programs in REX. *IEE Software Engineering Journal*, 7(2):73–82.

Logica, 1989. MacCadd Version 5.0—A new generation of CASE for the apple macintosh, Logica UK Ltd.

Magee, J., Kramer. J., and Sloman, M. 1989. Constructing distributed systems in conic. *IEEE Transactions on Software Engineering*, SE-15(6):663–675.

Magee, J., Dulay, N., and Kramer, J. 1993. Structuring parallel and distributed programs. *IEE Software Engineering Journal*, 8(2):73–82.

Magee, J., Dulay, N., and Kramer, J. 1994. Regis: A constructive development environment for distributed programs. *Distributed Systems Engineering Journal*, 1(5):304–312.

Martin, C.F. 1988. Second-generation CASE tools: A challenge to vendors. *IEEE Software*, 5(2):46–49.

Newton, P. and Browne, J.C. 1992. The CODE 2.0 graphical parallel programming language, Presented at *ACM Int. Conf. on Supercomputing*.

ObjecTime, 1993. *ObjecTime Overview*. ObjecTime Limited OT-R410V40-113.

Rich, C. and Waters, R. 1988. Automatic programming: Myths and prospects. *IEEE Computer*, 21(8):40–51.

Stovsky, M.P. and Weide, B.W. 1987. STILE: A graphical design and development environment. Presented at *COMPCON*, San Francisco.

Sunderam, V.S. 1990. PVM: A framework for parallel distributed computing. *Concurreny-Practice and Experience*, 2(4):315–339.

Wasserman, A.I. and Pircher, P.A. 1987. A graphical, extensible integrated environment for software development. *ACM SIGPLAN Notices (Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments)*, 22(1):131–142.

Automated Software Engineering 3, 285–307 (1996)
© 1996 Kluwer Academic Publishers. Manufactured in The Netherlands.

A Knowledge-Based Software Engineering Environment for Reusable Software Requirements and Architectures

H. GOMAA, L. KERSCHBERG, V. SUGUMARAN, C. BOSCH, I. TAVAKOLI AND L. O'HARA
Center for Information Systems Integration and Evolution, Department of Information and Software Systems Engineering, George Mason University, Fairfax, Virginia, 22030-4444

Abstract. This paper describes a prototype Knowledge-Based Software Engineering Environment used to demonstrate the concepts of reuse of software requirements and software architectures. The prototype environment, which is application-domain independent, is used to support the development of domain models and to generate target system specifications from them. The prototype environment consists of an integrated set of commercial-off-the-shelf software tools and custom developed software tools.

The concept of reuse is prevalent at several levels of the domain modeling method and prototype environment. The environment itself is domain-independent thereby supporting the specification of diverse application domain models. The domain modeling method specifies a family of systems rather than a single system; features characterize the variations in functional requirements supported by the family and individual family members are specified by the features they are to support. The knowledge-based approach to target system generation provides the rules for generating target system specifications from the domain model; target system specifications, themselves, may be stored in an object repository for subsequent retrieval and reuse.

Keywords: software engineering environments, software reuse, software architecture, knowledge-based software engineering, domain modeling

1. Introduction

The goal of large scale software reuse remains elusive in spite of efforts made during the past few years and apart from certain specific domains such as mathematical libraries. Most work in software reuse has addressed composition technology (Biggerstaff and Richter, 1987) where components are considered to be predominantly atomic and ideally unchanged when reused, although some adaptation may be required. Deriving new software systems from existing ones involves composition, where the components are the building blocks used in constructing the new system. This approach necessitates a library of reusable components and some approach for indexing, locating and distinguishing among similar components (Prieto-Diaz and Freeman, 1987). Problems with the reuse by composition approach include managing the large number of components a reuse library is likely to contain, and the difficulty in distinguishing among similar though not identical components. Having located and selected a component from the library, it is then the designer's responsibility to determine how this component fits into the new system.

This paper describes an approach that attempts to overcome these problems by taking an application domain perspective within the context of an evolutionary development life cycle. At George Mason University, a project is underway to support software engineering lifecycles, methods, and environments to support software reuse at the requirements and design phases of the software lifecycle in addition to the coding phase (Biggerstaff and Richter, 1987). A reuse-oriented software lifecycle, the Evolutionary Domain Lifecycle (Gomaa et al., 1989; Gomaa and Kerrchberg, 1991), has been proposed, which is a highly iterative lifecycle that takes an application domain perspective allowing the development of families of systems. A domain analysis and modeling method has also been developed (Gomaa, 1992a; Gomaa, 1993a) and applied to several application domains including NASA's Payload Operations Control Center (POCC) domain. After giving an overview of the method, this paper describes a prototype Knowledge-Based Software Engineering Environment, which has been developed to demonstrate the concepts of reusable software requirements and architectures. Both the method and prototype environment are illustrated with examples from NASAs POCC domain.

2. Domain modeling

2.1. Evolutionary domain life cycle

The Evolutionary Domain Life Cycle (EDLC) Model (Gomaa and Kerschberg, 1991) is a highly iterative software life cycle model that eliminates the traditional distinction between software development and maintenance. Furthermore, because new software systems are often outgrowths of existing ones, the EDLC model takes an application domain perspective facilitating the development of families of systems (Parnas, 1979). The EDLC consists of the following major activities (figure 1):

1. **Domain modeling.** Domain modeling refers to the development of reusable requirements, a reusable specification and a reusable architecture for the family of systems that constitute the application domain. Domain specific reusable components are developed and stored in an object repository.
2. **Target system generation.** Given the requirements of an individual target system (one of the members of the family), the target system specification is generated by tailoring the reusable specification and the target system architecture is generated by tailoring the reusable architecture. The component types to be included in the target system are selected based on the target system architecture. The concept of generating target systems from a generic specification and/or architecture has been investigated by several researchers (Batory, 1989; Batory and O'Malley, 1992; Kang et al., 1990; Pyster, 1990; Lubars, 1989).

2.2. Domain modeling method

2.2.1. Overview. In this paper, the emphasis is on domain modeling at the analysis phase. A Domain Model is a multiple view object-oriented model, also referred to as a problem-oriented architecture, for the application domain that reflects the common aspects and variations among the members of the family of systems that constitute the domain.

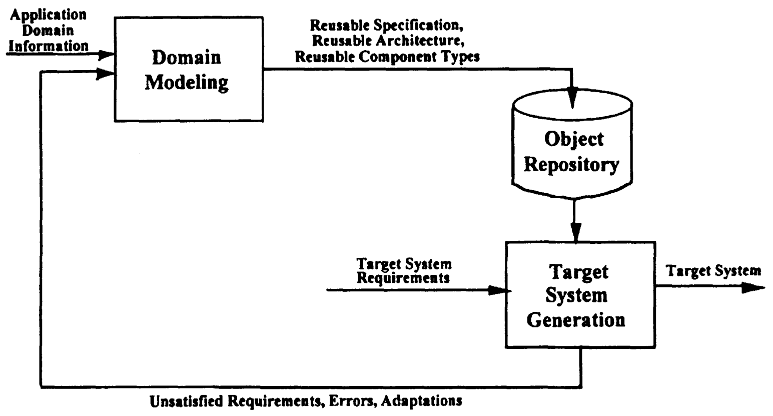

Application Domain Information
Domain Modeling
Reusable Specification, Reusable Architecture, Reusable Component Types
Object Repository
Target System Requirements
Target System Generation
Target System
Unsatisfied Requirements, Errors, Adaptations

Figure 1. Evolutionary domain life cycle model.

Since it is considered that the object-oriented model of software development is more conducive to evolution and change, the domain modeling approach takes an object-oriented perspective. The goal is to apply object-oriented concepts and extend them to application domains.

The domain modeling method is similar to other object-oriented methods when used for analyzing and modeling a single system (e.g., Rumbaugh et al., 1991; Schlaer and Mellor, 1988). Its novelty is the way it extends object-oriented methods to model families of systems. The method allows the explicit modeling of the similarities and variations in a family of systems.

In a domain model, an application domain is represented by means of multiple views, such that each view presents a different perspective on the application domain. This modeling approach is in contrast to Telos, which is a language-oriented approach for defining multiple view information systems (Mylopoulos et al., 1990).

Four of the views, the aggregation hierarchy, the object communication diagrams, the generalization/specialization hierarchy, and the state transition diagrams have similar counterparts in other object-oriented methods used for modeling single systems. However, in our domain modeling method, the aggregation hierarchy is also used to model *optional* object types, which are used by some but not necessarily all members of the family of systems. Furthermore, the generalization/specialization hierarchy is also used to model *variants* of an object type, which are used by different members of the family of systems. The fifth view, the *feature/object dependency view*, is used to represent explicitly the variations captured in the domain model; each feature is associated with the optional and variant object types needed to support it. This provides the basis for defining which target systems can be generated from the domain model.

The multiple views in the domain modeling method (Gomaa, 1992) are described below, with examples from the NASA Payload Operations Control Center Domain (POCC). The POCC application domain comprises the family of ground station command and control systems for unmanned satellites, which process and display satellite telemetry data and send commands up to the satellites.

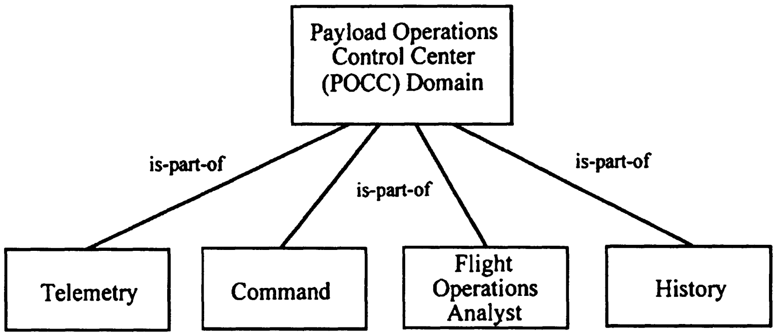

Payload Operations Control Center (POCC) Domain
is-part-of
is-part-of
is-part-of
Telemetry
Command
Flight Operations Analyst
History

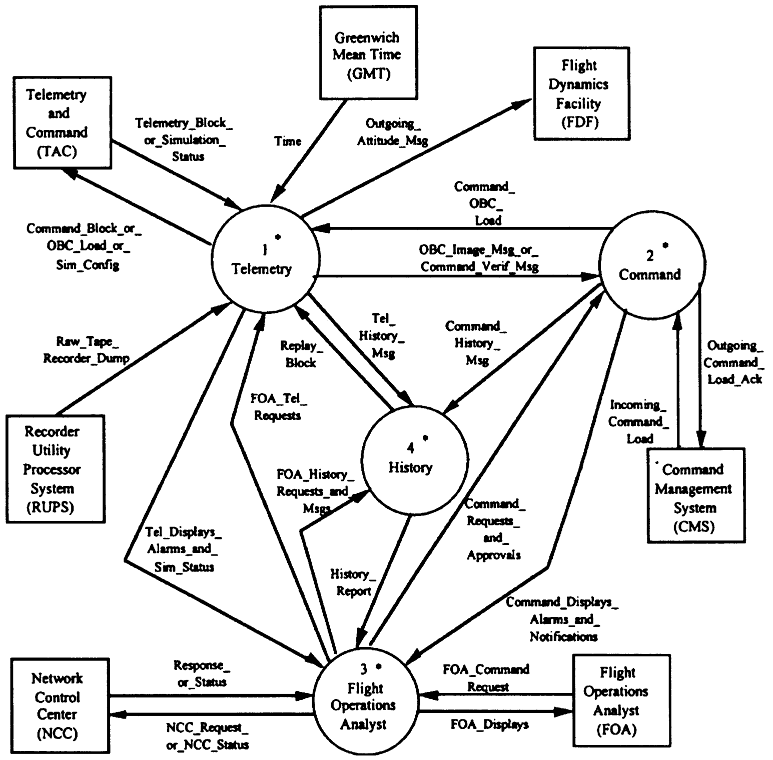
Figure 2. Aggregation hierarchy from payload operations control center domain.

2.2.2. Aggregation hierarchy. The Aggregation Hierarchy (AH) is used to decompose complex aggregate object types into less complex component object types eventually leading to simple object types at the leaves of the hierarchy. Object types are kernel, i.e., required in all target systems, or optional, only required in some target systems. At the upper levels of the hierarchy, aggregate object types represent subsystems, while the leaves of the hierarchy contain simple object types.

An example of the Aggregation Hierarchy for the Payload Operations Control Center Domain (POCC) is shown in figure 2. This whole domain is modeled as one aggregate object type called Payload Operations Control Center Domain. Payload Operations Control Center Domain contains four aggregate object types, which represent the four major subsystems of this application domain. These are Telemetry, Command, Flight Operations Analyst, and History. Each of these four subsystems in turn contains further aggregate or simple object types.

2.2.3. Object communication diagrams. Object types in the real world are modeled as concurrent tasks (Jackson, 1983), which communicate with each other using messages. The message interface between object types may be loosely coupled or tightly coupled (Gomaa, 1993b). The object communication diagrams (OCDs), which are hierarchically structured, show how object types communicate with each other. The levels of decomposition of OCDs correspond to the levels of the Aggregation Hierarchy. An example object communication diagram from the POCC domain is shown in figure 3.

The top level Object Communication Diagram for the POCC domain is shown in figure 3. In this figure, the bubbles represent object types, the boxes represent external entities, and the arcs represent message interfaces. The * in figure 3 means that these aggregate object types are decomposed further. As the decomposition of the Object Communication Diagrams corresponds to that of the Aggregation Hierarchy, figure 3 contains the same four subsystems as figure 2. In addition, figure 3 shows the message communication among the four subsystems, as well as to and from the external entities. Incoming telemetry data from the spacecraft comes from the Telemetry and Command entity to the Telemetry subsystem. GMT provides a timestamp. Recorded telemetry data is also input from Recorder Utility Processor System (RUPS).



Greenwich Mean Time (GMT)
Flight Dynamics Facility (FDF)
Telemetry and Command (TAC)
Telemetry_Block_ or_Simulation_ Status
Time
Outgoing_ Attitude_Msg
Command_ OBC_ Load
Command_Block_or_ OBC_Load_or_ Sim_Config
1 Telemetry
OBC_Image_Msg_or_ Command_Verif_Msg
2 Command
Raw_Tape_ Recorder_Dump
Replay_ Block
Tel_ History_ Msg
Command_ History_ Msg
Outgoing_ Command_ Load_Ack
FOA_Tel_ Requests
Incoming_ Command_ Load
Recorder Utility Processor System (RUPS)
4 History
Command Management System (CMS)
FOA_History_ Requests_and_ Msgs
Command_ Requests_ and_ Approvals
Tel_Displays_ Alarms_and_ Sim_Status
History_ Report
Command_Displays_ Alarms_and_ Notifications
Network Control Center (NCC)
Response_ or_Status
3 Flight Operations Analyst
FOA_Command Request
Flight Operations Analyst (FOA)
NCC_Request_ or_NCC_Status
FOA_Displays

Figure 3. Level 0 object communication diagram from payload operations control center domain.

The Telemetry subsystem is responsible for processing the telemetry data received from the satellite, which includes converting it to engineering units and generating alarms for sensor values that are out of range, storing current values of the raw and processed telemetry data, and sending this data to the History subsystem for archival storage. The Telemetry subsystem also responds to requests for current telemetry and alarm data from the Flight Operations Analyst (FOA) User Interface subsystem, which displays it to the analyst. The FOA may also request to view historical telemetry data from the History subsystem. In addition, the FOA can prepare spacecraft commands, referred to as real-time commands, which are passed on to the Command Subsystem. The Command Subsystem also receives command loads from an external subsystem, the Command Management System. The Command Subsystem sends command loads and real-time commands to the Telemetry Subsystem for uploading to the satellite; it also monitors the progress of command execution on board the satellite.

2.2.4. State transition diagrams. As each object type is modeled as a sequential task, an object type may be defined by means of a finite state machine, and represented by a state transition diagram, whose execution is by definition strictly sequential.

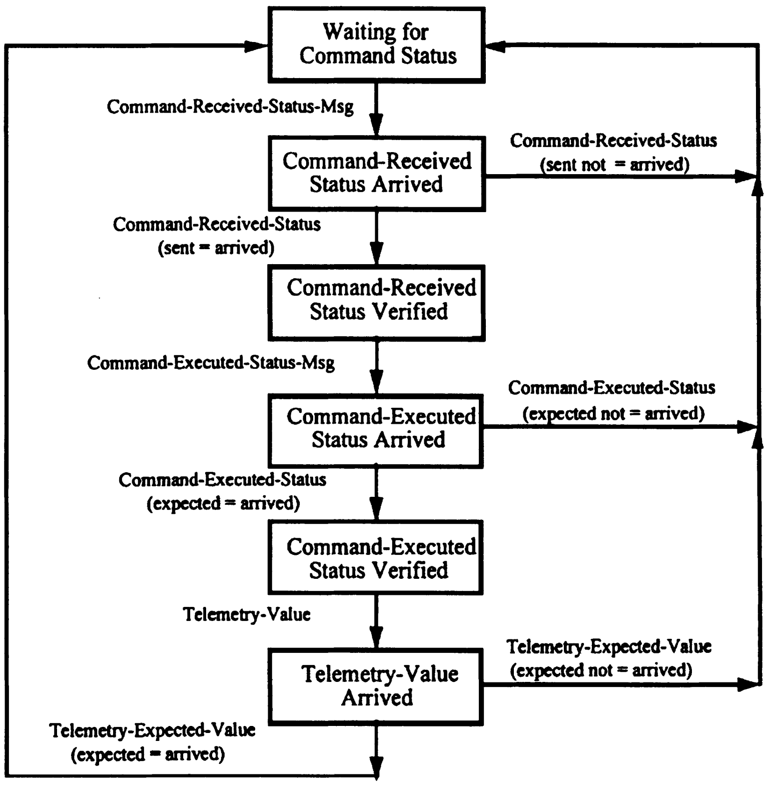

Waiting for Command Status
Command-Received-Status-Msg
Command-Received Status Arrived
Command-Received-Status (sent not = arrived)
Command-Received-Status (sent = arrived)
Command-Received Status Verified
Command-Executed-Status-Msg
Command-Executed Status Arrived
Command-Executed-Status (expected not = arrived)
Command-Executed-Status (expected = arrived)
Command-Executed Status Verified
Telemetry-Value
Telemetry-Value Arrived
Telemetry-Expected-Value (expected not = arrived)
Telemetry-Expected-Value (expected = arrived)

Figure 4. State transition diagram from payload operations control center domain.

An example state transition diagram from the POCC domain is given for the Earthbound Command Load Verifier object type in figure 4. When a command is sent up to the satellite to be executed, there are several states for verifying that the command was executed as expected. First a command received status message needs to be checked to determine that the command did arrive at the satellite, next a command executed status message is checked to determine that the command was executed. Finally, the appropriate telemetry data is checked against an expected value to determine whether the command produced the desired result. If any of these checks fails to yield the expected result, the command verification is aborted.

2.2.5. Generalization/specialization hierarchies. As the requirements of a given kernel or optional object type are changed to meet the specific needs of a target system, the object type may be specialized (Meyer, 1987). The variants of a domain object type are specified in a Generalization/Specialization Hierarchy (GSH).

An example Generalization/Specialization Hierarchy from the POCC domain is shown in figure 5 for the Observatory Instrument Telemetry Analyzer object type. This object type monitors an onboard satellite instrument. There are several variants of this object type corresponding to different onboard observatory experiments.

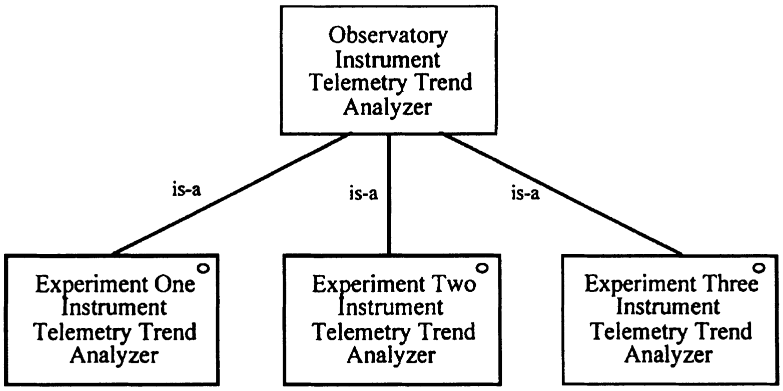

Observatory Instrument Telemetry Trend Analyzer
is-a
is-a
is-a
Experiment One Instrument Telemetry Trend Analyzer
Experiment Two Instrument Telemetry Trend Analyzer
Experiment Three Instrument Telemetry Trend Analyzer

Figure 5. Generalization/specialization hierarchy from payload operations control center domain.

2.2.6. Feature/object dependencies. For each feature (domain requirement), this view shows the object types required to support the feature. In domain analysis, domain requirements are analyzed and categorized as those that support kernel requirements (must be supported in all target systems), optional requirements (only required in some target systems), prerequisite requirements (depended upon by other requirements), and those that are mutually exclusive. This view emphasizes optional features, because it is the selection of the optional features and the object types required to support them that determine the nature of the desired target system.

Examples of feature/feature and feature/object dependencies from the POCC domain are shown in figure 6. Consider the feature/object dependencies for the Sending Real-Time commands feature. Sending command loads, i.e., predetermined sequences of commands to be executed on the satellite at predetermined times, is a kernel feature. Thus every POCC system must support this feature. However sending real-time commands, which are commands entered on-line by the Flight Operation Analyst, for execution as soon as they are received by the satellite, is an option that only some POCC systems support. The object types required to support this feature are the Real-Time Command Data Store (which maintains a copy of the command after it has been sent), Satellite Bound Command Problem Resolver (which resolves any inconsistencies between a real-time command and a command

Example of feature/object dependencies:

(Sending Real Time Commands Feature **supported-by** Real-Time_Command-
Data_Store_OS optional)
(Sending Real Time Commands Feature **supported-by** Satellite_Bound-
Command_Problem_Resolver_OS optional)
(Sending Real Time Commands Feature **supported-by** Satellite_Bound_Real-
Time_Command_Processor_OS optional)

Example feature/feature dependency :

(Verifying Real Time Commands Feature **requires** Sending Real Time Commands Feature)

Figure 6. Example feature/object and feature/feature dependencies from payload operations control center domain.

in the regular command load), and Satellite Bound Real-Time Command Processor, (which processes all real-time commands to be sent to the satellite). These object types are only included in a given target POCC system (one of the members of the POCC family) if the sending real-time commands feature is needed for that system.

An example of a feature/feature dependency is the Verifying Real-Time Commands feature, which contains the functionality for determining that real-time commands were executed as expected on the satellite. This optional feature needs the Sending Real-Time Commands as a prerequisite, as it is meaningless to verify real-time commands if they were never sent to the satellite in the first place.

3. Overview of prototype knowledge-based software engineering environment

3.1. Domain modeling environments

A software engineering environment which allows the generation of target systems from a domain model, i.e., reusable architecture for the particular application domain, is referred to in this paper as a domain modeling environment. A goal of this domain modeling environment research is to be application domain-independent. This is in contrast to generator environments such as application generators and software system generators (Batory, 1989; Batory and O'Malley, 1992). These generators are usually highly domain-specific as they have the structure and code for the application domain built into them. In addition, they provide a means of adapting the code to generate a specific target system, either by means of parameterization or a by a user program written in a domain-specific language. The most widespread use of this technology is in application generators and fourth generation languages, where the application domain is that of interactive database-intensive information systems (Blum, 1987). A user writes a program in the fourth generation language, which is used by the application generator to generate a specific target system.

3.2. Objectives of prototype

The EDLC and the domain modeling concept of developing a family of systems represent a radically different paradigm for software development compared to the traditional development paradigm of developing a single system. It was therefore considered desirable to develop a proof-of-concept prototype domain modeling environment. The prototype environment is called the Knowledge-Based Software Engineering Environment (KBSEE).

The objectives of the proof-of-concept prototype environment are to:

(a) Provide tool support for representing the multiple graphical views supported by the domain modeling method.
(b) Provide a capability for consistency checking between the multiple views.
(c) Provide a capability for mapping the multiple views to a common underlying representation, namely an object repository.

(d) Provide automated support for generating target system specifications from the domain model.
(e) Provide a domain independent environment. Thus the KBSEE should be capable of being used with multiple domain models.
(f) Because of limited resources and the need to focus those resources on the innovative parts of the KBSEE, use existing software tools where possible.

At the start of this project, these objectives were considered daunting and limited resources were available for this purpose. It was therefore decided to constrain the proof-of-concept experiment as follows:

(a) From domain modeling, provide tool support for the domain analysis and modeling phase of the EDLC.
(b) From target system generation, provide tool support for the generation of the target system specification phase. In particular, it was viewed that this phase was a good candidate for a knowledge-based tool, as the procedures for target system generation could be expressed as rules.

3.3. Tool support for prototype

Because of limited resources and the desire to focus on the innovative aspects of the domain modeling environment, commercial-off-the-shelf software was used whenever possible. We believe that the tools discussed below represent another important aspect of reuse, and feel that open architectures and standards are essential to fostering large-scale reuse of software.

User interface management system. An existing user interface management system is desirable to support a windows, menu and icon based user interface. NASA's TAE User Interface Management System was selected to provide an integrated interface to the prototype environment (Szczur, 1990). The main menu of the environment is built using TAE, as is the user interface for the object repository tools described in Section 4.3.

CASE tool. A survey of CASE tools indicated that there are several that support the popular Structured Analysis and Real-Time Structured Analysis methods (Yourdon, 1989). It was decided that the domain modeling method should use a graphical notation similar to that used by Real-Time Structured Analysis, but with a radically different semantic interpretation of the diagrams supported by the domain modeling method. Another key requirement was that the CASE tool support an open system architecture, so that the information contained in the multiple views could be extracted and processed by custom software tools developed as part of this project. Interactive Development Environment's (IDE) Software Through Pictures (StP) was selected, as it satisfies the above two requirements. The StP CASE tool was used by the Domain model graphical editing tools described in Section 4.2.1.

Knowledge based expert system shell. A Knowledge-Based Requirements Elicitation Tool was needed for automatically generating target system specifications from the domain model. Using an expert system shell would greatly assist in the development of this tool. NASA's CLIPS expert system shell was selected for this purpose. The Knowledge-Based Requirements Elicitation Tool (KBRET), described in Section 4.4, was developed using CLIPS.

Object-oriented programming environment. Initially, we considered using an object-oriented database management system as the basis for the object repository. However, for the proof-of-concept prototype, it was decided that it would be simpler and sufficient to implement the object repository using the Eiffel object-oriented programming language and system (Meyer, 1987), which supports a persistent object store. The object repository tools described in Section 4.3 were all developed using the Eiffel programming environment.

4. The knowledge-based software engineering environment

4.1. Introduction

The scope of the prototype Knowledge-Based Software Engineering Environment includes two major phases: (1) Development of a domain model specification and (2) Generation of the target system specification. The integrated underlying representation of the domain model, as captured by the KBSEE, is referred to as the domain model specification, i.e., the specification for the family of systems. A tailored version of the domain model specification is referred to as a target system specification, i.e., the specification of a member of the family.

During domain modeling (figure 7), the graphical editors supported by the Software Through Pictures CASE tool are used to develop four of the multiple views of the domain model, namely the Aggregation Hierarchy, the Object Communication Diagrams, the Generalization/Specialization Hierarchies and the State Transition Diagrams. The information in the multiple views is extracted, checked for consistency, and mapped first to a set of relations and then to an object repository. The feature/object dependencies and feature/feature dependencies are defined using a Feature/Object Editor. The tools involved in this phase are described in Sections 4.2 and 4.3.

A knowledge based requirements elicitation tool (KBRET) is used to assist with the generation of the target system specification (figure 8). KBRET conducts a dialog with the human target system requirements engineer, presenting the user with the optional features available for selection for the target system. The user selects the desired features, and KBRET reasons about the feature/feature dependencies to ensure that a consistent set of features are selected. KBRET then determines the kernel, optional and variant object types to be included in this target system. This output of KBRET is used to adapt the multiple views of the domain model to generate the multiple views of the target system specification. The tools involved in this phase are described in Sections 4.4 and 4.5.

Apart from the Domain Model Graphical Editing tools described in Section 4.2.1, the tools described in this section were custom developed for this project.

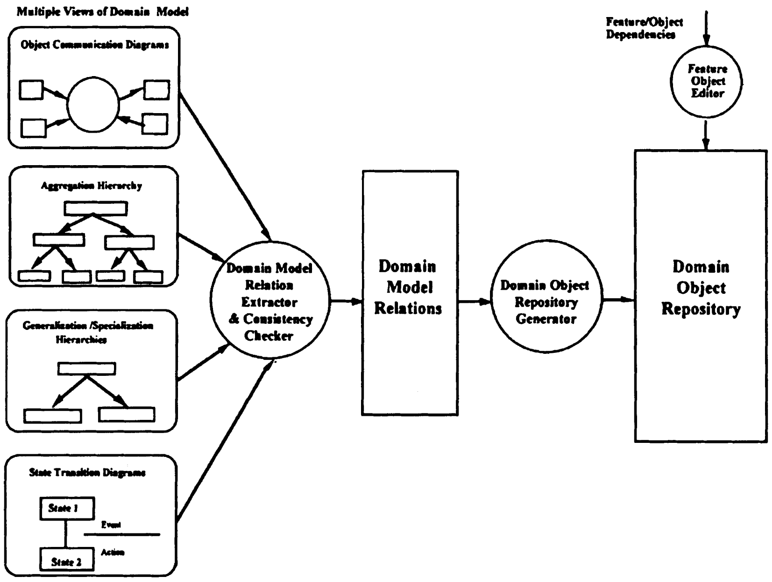

Multiple Views of Domain Model
Object Communication Diagrams
Aggregation Hierarchy
Generalization /Specialization Hierarchies
State Transition Diagrams
State 1
Event
Action
State 2
Domain Model Relation Extractor & Consistency Checker
Domain Model Relations
Domain Object Repository Generator
Feature/Object Dependencies
Feature Object Editor
Domain Object Repository

Figure 7. Development of domain model specification.

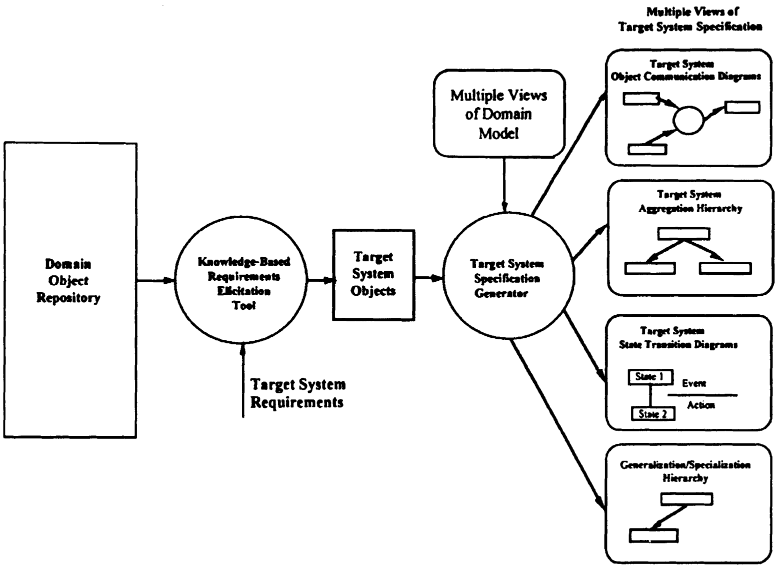

Multiple Views of Target System Specification
Target System Object Communication Diagrams
Multiple Views of Domain Model
Domain Object Repository
Knowledge-Based Requirements Elicitation Tool
Target System Objects
Target System Specification Generator
Target System Aggregation Hierarchy
Target System State Transition Diagrams
State 1
Event
Action
State 2
Generalization/Specialization Hierarchy
Target System Requirements

Figure 8. Generation of target system specification.

4.2. *Tools for creating and integrating multiple views*

In this section we discuss each custom-developed software tool.

4.2.1. Domain model graphical editing tools. These tools allows the graphical editing of four of the five multiple views of a domain model. The views are the object type aggregation hierarchy, the object communication diagrams, state transition diagrams, and object type generalization/specialization hierarchies. These tools are customized versions of the Software through Pictures (StP) CASE tool graphical editors.

The Aggregation Hierarchy (figure 2) is developed using StP's Structure Editor. Object Communication Diagrams (figure 3) are developed using StP's data flow diagram editor. State transition diagrams (figure 4) are developed using StP's state transition diagram editor. The Generalization/Specialization Hierarchies (figure 5) are also created using the StP structure editor; the *o* in the corner of the box is used to indicate that the children in the hierarchy are variants of the parent object type, and hence distinguishes the Generalization/Specialization Hierarchies from the Aggregation Hierarchy.

4.2.2. Domain model relation extractor tool. The information contained in the four views of the domain model described above, as captured by StP's graphical editors, are stored by StP in its TROLL relational data base. The Domain Model Relation Extractor (figure 7) is a custom developed tool that extracts this information from StP's database, interprets the data semantically according to the domain modeling method, and stores this data in a common underlying relational representation of the multiple views.

As mentioned previously, the domain modeling method's semantic interpretation of the multiple views differs from the StP interpretation. The StP underlying relational schema was expanded by adding a new set of relations that captured the semantics of the domain model. The Domain Model Relation Extractor (DMRE) tool uses a set of scripts written in the Troll/USE query language to: (1) extract the domain information from the predefined set of relations, (2) interpret them semantically based on the domain modeling method, and (3) store the extracted information in the newly defined Domain Model Relations.

There is one relation for the aggregation hierarchy, one relation for the generalization/specialization hierarchies, and several relations to capture the information contained in the object communication diagrams. The Domain Model Relations are:

(1) The aggregation hierarchy is captured in the Node_part_of relation. The attributes of this relation are the parent node name and the child node name.
(2) The generalization/specialization hierarchies are captured in the Is_a relation. The attributes of this relation are the parent node name and the child node name.

The information in the object communication diagrams is captured in the following relations:

(1) The domain object types are captured in the Nodes relation. The attributes of this relation are the name of the object type, the name of the diagram it appears on (each diagram has a unique name), its unique index which is displayed on the diagram, and its cardinality.

(2) The Arcs relation defines the message interfaces shown on the object communication diagrams. The attributes of this relation are the message name, the diagram it appears on, and the source and sink object types.
(3) The Externals relation defines the names of the external object types that appear on the domain context diagram. This is the highest level object communication diagram, which defines the scope of the domain.
(4) The Arc_part_of relation defines the aggregate message decomposition as depicted on the object communication diagrams. The attributes of this relation are the parent arc label and the child arc label.
(5) The Decomposed relation defines the decomposition of aggregate object types into their constituent object types as given by the names of the parent and child object communication diagrams. The attributes of this relation are the parent node name, the parent OCD diagram name, and the child OCD diagram name.
(6) The Diagrams relation defines the names of all the object communication diagrams.

As an example of the Domain Model Relations, a portion of the Node_part_of relation from the POCC application domain is shown in figure 9. This relation has two attributes corresponding to the parent and child nodes. For every parent-child relationship in the aggregation hierarchy, there is a corresponding tuple in the Node_part_of relation. Thus if a parent has several children, such as Command which has three children, there is one row for each child. As multilevel hierarchies are supported, a child in one row can appear as a parent in a different row.

The notation suffixes K, O, A, S, and V refer to characteristics of the object types, respectively, Kernel, Optional, Aggregate, Simple and Variant. An object type is either Kernel or Optional, Aggregate or Simple, and may also be a Variant. These labels indicate the possibly multiple roles an object type may have in the domain model.

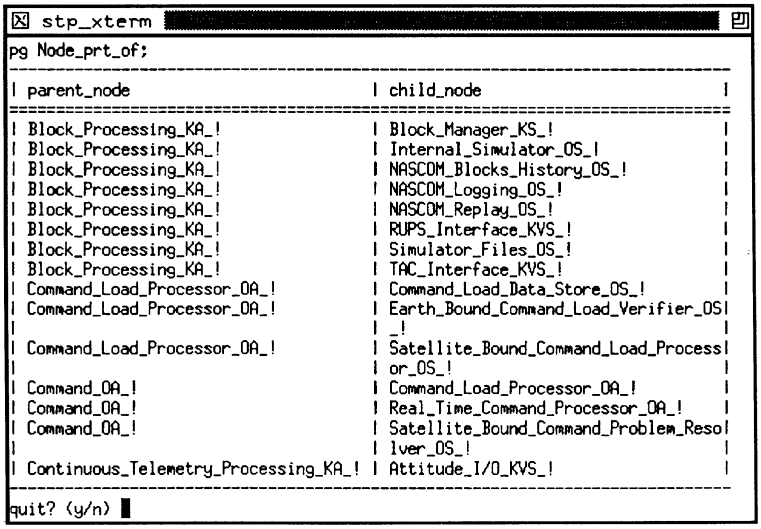

```
stp_xterm
pg Node_prt_of;
-------------------------------------------------------------------------
| parent_node                          | child_node                       |
=========================================================================
| Block_Processing_KA_!                | Block_Manager_KS_!               |
| Block_Processing_KA_!                | Internal_Simulator_OS_!          |
| Block_Processing_KA_!                | NASCOM_Blocks_History_OS_!       |
| Block_Processing_KA_!                | NASCOM_Logging_OS_!              |
| Block_Processing_KA_!                | NASCOM_Replay_OS_!               |
| Block_Processing_KA_!                | RUPS_Interface_KVS_!             |
| Block_Processing_KA_!                | Simulator_Files_OS_!             |
| Block_Processing_KA_!                | TAC_Interface_KVS_!              |
| Command_Load_Processor_OA_!          | Command_Load_Data_Store_OS_!     |
| Command_Load_Processor_OA_!          | Earth_Bound_Command_Load_Verifier_OS|
|                                      | _!                               |
| Command_Load_Processor_OA_!          | Satellite_Bound_Command_Load_Process|
|                                      | or_OS_!                          |
| Command_OA_!                         | Command_Load_Processor_OA_!      |
| Command_OA_!                         | Real_Time_Command_Processor_OA_! |
| Command_OA_!                         | Satellite_Bound_Command_Problem_Reso|
|                                      | lver_OS_!                        |
| Continuous_Telemetry_Processing_KA_! | Attitude_I/O_KVS_!               |
-------------------------------------------------------------------------
quit? (y/n)
```

Figure 9. Node_part_of relation from POCC domain.

4.2.3. Domain model consistency checking tool. The graphical views represented by the OCDs, the AH, the GSHs, and the STDs, each focus on one aspect of the domain being modeled. Although StP provides consistency checking within one particular view, additional consistency checking is required among the multiple views.

The Domain Model Consistency Checker (DMCC) (figure 7) uses a set of scripts written in the Troll/USE query language that check the underlying relations for inconsistencies based on rules described by the domain modeling method outlined in Section 2.2. The rules that are checked are the following:

(1) There should be a one-to-one correspondence between the object types in the *i*th level OCD and the object types in the corresponding level in the AH.
(2) The root node in each GSH should correspond to a leaf node in the AH. This is due to the fact that each GSH serves as the specialization of a leaf object type in the AH.
(3) For each active leaf object type in the OCDs, there must exist a state transition diagram that captures the internal behavior of the object type. Each state transition diagram, on the other hand, must have a corresponding active leaf object type in the OCDs.
(4) The events in each state transition diagram (STD) should correspond to the incoming messages of the object type in the OCD that the STD is describing. The actions in each STD, on the other hand, should correspond to the outgoing messages of the same object type in the OCD.

The domain modeler runs the DMCC after the multiple views have been developed. DMCC displays any inconsistencies that the domain modeler must then correct before proceeding to the generation of the object repository.

4.3. Object repository

The object repository provides an integrated object-oriented specification of each object type in the multiple views of the domain model. This repository is a single composite object that is composed of other objects representing domain object types, features, and the relationships among them which serve to define a domain model. This section gives a brief description of the object repository. A more detailed description is given in (Bosch, Gomaa, and Kerschberg, 1995).

4.3.1. Domain object repository generator. The Domain Object Repository Generator tool (figure 7) takes the information captured in the relational representation and creates corresponding objects according to the object repository's schema. For example, if the domain analyst had created eight object communication diagrams using StP, the Domain Object Repository Generator tool would create eight instances of class OCD, the class defining object communication diagrams. Similarly, this tool would create objects representing the aggregation hierarchy, generalization/specialization hierarchies, and state transition diagrams, as well as the domain object types, external object types, and messages which are represented in these diagrams.

4.3.2. Feature/object editor. After the object repository representing a domain model has been created, the domain analyst can use the feature/object editor (figure 7) to define the optional features by: (1) giving each feature a unique name, (2) entering an informal annotation for each feature, (3) specifying domain object types supporting the feature being defined, and (4) specifying other prerequisite features required by the feature being defined. In addition to defining new features, the domain analyst can use this tool to browse features previously defined for a given domain model, delete features from the domain model, or modify the definition of features in a domain model. The Feature/Object Editor can also be used to establish relationships among sets of features.

4.3.3. Domain-dependent knowledge base extractor. The Domain-Dependent knowledge base Extractor tool extracts the information contained in the object repository for a domain model and maps it to the domain-dependent knowledge base, which contains a knowledge-based representation of the information contained in the multiple views. This knowledge base is stored as facts in the CLIPS language. Facts are created for each object type, feature, and feature annotation. Similarly, facts are created corresponding to feature/feature dependencies, feature/object dependencies, as well as the aggregation and generalization/specialization hierarchies. An example of a domain-dependent knowledge base for the POCC application domain is shown in figure 10. This figures shows a selection

Object
(Object: 0 Payload_Operations_Control_Center_Domain_KA kernel aggregate agh_root)
(Object: 1 Telemetry kernel aggregate)
(Object: 2 Telemetry Pre-Processor kernel)
• • •
(Object: 65 POCC Mode Selector With Simulation variant)

Features
(Feature: 1 Data Collection of Simulated Telemetry)
• • •
(Feature: 7 Verifying Real Time Commands)

Feature/Feature Dependencies
(Verifying Real Time Commands Feature 7 requires Sending Real Time Commands Feature 4)

Feature/Object Dependencies
(Data Collection of Simulated Telemetry Feature 5 supported-by 65 POCC Mode Selector With
Simulation variant)
(Sending Real Time Commands Feature 4 supported-by 38 Real-Time Command Data
Store optional)
• • •
(Verifying Real Time Commands Feature 7 supported-by 26 Earth Bound Real-Time Command
Verifier optional)

Aggregation Hierarchy
(is-part-of 0 Payload Operations Control Center Domain 1 Telemetry)
(is-part-of 1 Telemetry 3 Spacecraft Telemetry Processor)
(is-part-of 3 Spacecraft Telemetry Processor 5 SC Engineering Telemetry Trend Analyzer)
• • •

Generalization/Specialization Hierarchies
(is-a 10 Observatory Instrument Telemetry Analog Limits Checker 53 Experiment One
Instrument Telemetry Analog Limits Checker)
(is-a 10 Observatory Instrument Telemetry Analog Limits Checker 55 Experiment Two
Instrument Telemetry Analog Limits Checker)
• • •

Figure 10. Domain-dependent knowledge base for the POCC.

of the domain object types, features, feature/feature dependencies, feature/object dependencies, and parts of the aggregation hierarchy and generalization/specialization hierarchies.

The characteristics of each object type in the domain model are also given. Thus object type Payload Operations Control Center Domain is the root of the Aggregation Hierarchy, which is always Kernel and Aggregate. Features are supported by variant or optional object types. A feature, such as Verifying Real Time Commands, may require another feature as a prerequisite.

4.4. Knowledge-based requirements elicitation tool

4.4.1. Overview. A target system specification is derived by tailoring the domain model according to the features desired in the target system. During the generation of the target system specification, the feature/object dependencies must be enforced in order to ensure a consistent specification. A knowledge-based system called the Knowledge-Based Requirements Elicitation Tool (KBRET) has been developed to automate the process of generating the specifications for the target systems. This tool has been implemented in NASA's CLIPS expert system shell (CLIPS, 1989).

The major components of KBRET are (1) the domain-dependent knowledge base, (2) the domain-independent knowledge base, and (3) the user interface manager. The domain-dependent knowledge base is derived from the object repository through the KBRET-Object Repository Interface, and contains domain-specific information which characterizes the multiple views, object specifications, features and dependencies of a particular domain model for which a target system is desired.

The domain-independent knowledge base contains the procedural and control knowledge required to generate target system specifications from a domain model. This separation between the domain-independent and domain-dependent knowledge is essential for making the environment domain independent. Thus, the domain-dependent knowledge base can be derived from different domain models regardless of their application domain. The knowledge bases consist of knowledge modules (KMs). Each domain-dependent KM consists of a set of related facts derived from the object repository, while each domain-independent KM consists of rules to support its functionality. The inference engine is the underlying forward-chaining production system provided by CLIPS. The KMs are invoked and executed by the inference engine, based on the rules in the domain-independent knowledge base.

KBRET accomplishes the task of target system specification generation in several phases: Browsing, Target System Requirements Elicitation, Dependency Checking, and Target System Specification Generation. The various components of KBRET are schematically shown in figure 11.

The User Interface Manager is responsible for interacting with the target system engineer to elicit the requirements for the target system. It addresses such issues as how, and in what sequence the target system engineer should be prompted for various features, as well as the invocation and control of the various KMs of KBRET.

4.4.2. Domain-dependent knowledge base. As the name suggests, the domain-dependent knowledge base contains specific information about a particular application domain

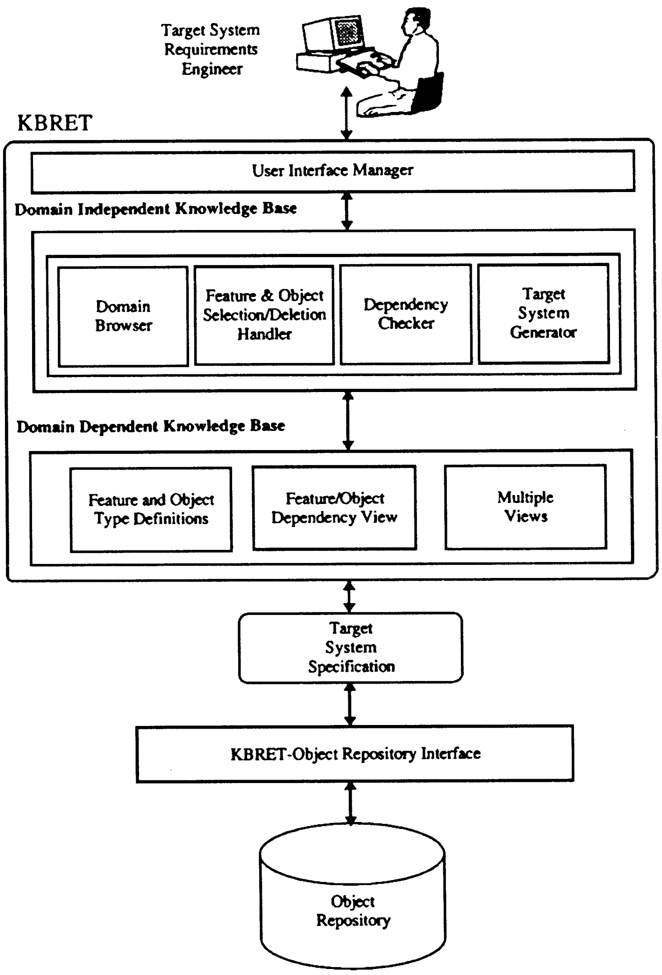

Target System Requirements Engineer
KBRET
User Interface Manager
Domain Independent Knowledge Base
Domain Browser
Feature & Object Selection/Deletion Handler
Dependency Checker
Target System Generator
Domain Dependent Knowledge Base
Feature and Object Type Definitions
Feature/Object Dependency View
Multiple Views
Target System Specification
KBRET-Object Repository Interface
Object Repository

Figure 11. Knowledge-based requirements elicitation tool.

(figure 10). This knowledge base is composed of several modules, namely, "Feature and Object Types", "Feature/Object Dependencies", and "Multiple Views". They are used by the domain-independent knowledge base of KBRET in eliciting the requirements and generating the target system specification. The domain-dependent knowledge base is derived from the domain model specification, stored persistently in the object repository.

The Features and Object Types KM contains a list of all the object types and features specified in the domain model. For each object type, its identifier, name, and properties are stored in this KM. The properties of object types are: kernel, optional, variant, aggregate, root of aggregation hierarchy, and root of generalization/specialization hierarchy. Similarly, for each feature, its identifier and name are stored. The various relationships and dependencies among features and between features and object types are captured in the

Feature/Object Dependencies KM. The Multiple Views KM contains the different views created using the domain modeling method, in particular, the aggregation hierarchy and the generalization/specialization hierarchies. These hierarchies are accessed and utilized by the Target System Generator KM when the target system is being assembled.

4.4.3. Domain-independent knowledge base. The domain-independent knowledge base provides procedural and control knowledge for the various functions supported by KBRET.

Before specifying the requirements for the target system, the target system engineer may wish to browse portions of the domain model to gain greater understanding of the application domain under consideration. The Domain Browser KM provides this facility. It contains rules for initiating and terminating the browsing session and also provides access to the appropriate domain-dependent KMs to be accessed in order to browse those parts of the domain model which the target system engineer wishes to explore.

The Feature & Object Selection/Deletion Handler KM keeps track of the selection or deletion of features, and associated object types. This KM incorporates rules for selecting and deleting features and also rules for checking feature/feature and feature/object dependencies.

The Dependency Checker KM works cooperatively with the Feature & Object Selection/Deletion Handler KM. Whenever a feature is selected or deleted, the Dependency Checker enforces the feature/feature and feature/object dependencies, which are obtained from the Feature/Object Dependencies KM. When a feature with some prerequisite features is selected, the Dependency Checker ensures that those prerequisite features are included in the target system.

In order to support the complex interrelationships among features and objects, constraint management techniques (Shepherd and Kerschberg, 1986) are used to reason about these relationships, and ensure consistency through rule inference and triggers to enact constraint propagation (Kerschberg, 1990; Yoon and Kerschberg, 1995).

For example, in the POCC application domain (figure 10), the "Verifying Real Time Commands" feature requires the "Sending Real Time Commands" feature. When selecting the Verifying Real Time Commands feature, if the "Sending Real Time Commands" feature is not selected, it will be selected automatically for the target system. Similarly, before deleting a feature from the target system, dependency checking is performed to ensure that it is not required by any other target system feature. For example, if both "Sending Real Time Commands" and "Verifying Real Time Commands" features are currently selected, the "Sending Real Time Commands" feature may not be deleted as long as the "Verifying Real Time Commands" feature is selected for the target system.

Once feature selection for the target system has been completed, the Target System Generator KM begins the process of assembling the target system. The domain kernel object types are automatically included in the target system. Depending upon the features selected for the target system, the corresponding variant and optional object types are included according to the feature/object dependencies.

When the target system assembly is complete, KBRET produces two relations: (1) the object types that have been included in the target system and (2) the specializations that have been included in the target system.

The generated target system specification is stored in the object repository for future reference and reuse. The target system specification can be reused because we store not only the specification, but also the features and the reasoning "state". KBRET can be used to make "incremental changes" to an existing target system by de-selecting certain features and selecting other features. A new target system specification can then be generated.

4.5. Target system specification generator tool

The graphical views of a target system can be generated automatically from those of the domain model by tailoring the domain model views based on KBRET's output. The specification of a target system is defined in terms of the object types that are to be included in the target system. Using this information, the Target System Specification Generator tool performs the following tasks:

(1) Derives the set of object types that are not included in the target system and hence must be removed from the domain model.
(2) Generates the graphical views for the target system using the domain model views and the list of the object types to be deleted. The two relations that KBRET generates are used in tailoring the StP "picture files" of the domain model to create the target system picture files to be displayed by StP. (StP creates a picture file for each diagram describing the pictorial layout of the diagram).
(3) Modifies the object type names by appending the word "Variant" to the name of those object types for which a specialization, has been selected and the word "Variants" to those object types for which more than one variant object type has been selected.

The target system engineer may then view the multiple views of the target system using the StP graphical editors.

5. Evaluation of KBSEE

The KBSEE is a successful proof-of-concept prototype. The KBSEE demonstrates the feasibility of domain-independent domain modeling methods and environments for developing domain models, which capture the common aspects and variations of a family of systems, from which target system specifications can be generated.

Specifically, each of the objectives listed in Section 3.2 was achieved as follows:

(a) Provide tool support for representing the multiple graphical views supported by the domain modeling method.

This was achieved using the StP graphical editors to support the multiple views. The editors were used to capture the domain model; each view was then interpreted semantically by our tools according to the domain modeling method.

(b) Provide a capability for consistency checking between the multiple views.

We developed a multiple view consistency checking tool for this purpose, which reported any inconsistencies among the views to the user.

(c) Provide a capability for mapping the multiple views to a common underlying representation, namely an object repository.

This was achieved by first using the open architecture provided by StP to extract the information in the multiple views, mapping these views to an integrated set of relations that supported the multiple views, and then mapping these relations to an object repository. The latter two steps were achieved using tools we developed for the KBSEE.

(d) Provide automated support for generating target system specifications from the domain model.

This was achieved by developing the knowledge based requirements elicitation tool (KBRET) for this purpose. KBRET interacts with the target system requirements engineer to generate a target system specification from the domain model.

(e) Provide a domain independent environment. Thus the KBSEE should be capable of being used with multiple domain models.

Several domain models have been developed using the KBSEE. In addition to NASA's Payload Operations Control Center domain described in this paper, other application domains have been modeled including NASA's Transportable Payload Operations Control Center (TPOCC) domain, a manufacturing domain (Gomaa, 1995a) and a banking federation domain (Gomaa, 1994).

This demonstrates that the KBSEE environment is indeed domain independent. Domain independence is achieved by treating all domain dependent information as data and facts to be manipulated by the domain-independent tools.

(f) Use existing software tools where possible.

As an experiment in constructing an environment by building on top of several tools and integrating these tools, the KBSEE was highly successful. It succeeded in integrating several tools including the StP CASE tool, the Eiffel programming environment, the TAE user interface management system, and the CLIPS expert system shell.

As shown above, the KBSEE is a successful proof-of-concept prototype which met all its objectives. However, because KBSEE was built as a proof-of-concept prototype, the emphasis was on demonstrating that the domain modeling concept was viable and not on providing an easy to use operational environment. Thus, from the point of view of forming the basis of an operational system for use by large numbers of users, KB-SEE has some limitations. These include inconsistencies in the user interface, as the StP editors have a different user interface than tools using TAE, which in turn are different from KBRET's user interface. Furthermore, some of the tools are intolerant to user errors. None of these limitations are difficult to address from a development perspective.

6. Current status

We are currently investigating extending the KBSEE to support the design and implementation phases of the EDLC model as well as scaleup issues. Current research is investigating a key aspect of defining the reuable architecture; defining the interconnection among the kernel, optional, and variant object types in the architecture using a module interconnection language. Object types and their interconnections are defined in object interconnection fragments, corresponding to the features that depend on them.

For each feature, a fragment is created, in which the optional and variant object types needed to support it are defined as well as the interconnection between these object types. In addition, interconnections are defined between these object types and any kernel object types they use, as well as any optional or variant object types defined in prerequisite features of this feature.

A given target system is defined in terms of the object interconnection fragments it needs. The target system always contains the kernel fragment. In addition, it contains the optional fragments corresponding to the optional features selected for that target system, subject to the appropriate feature/feature constraints.

We are currently interfacing the domain modeling environment to the Regis distributed configuration environment (Magee et al., 1994). Regis' flexible and comprehensive support for constructing distributed systems makes it a good vehicle for configuring distributed applications developed using the domain modeling method described in this paper. Particularly useful is the capability for configuring distributed applications from predefined component types.

A domain model is defined in terms of Regis component types stored in a reuse library. All component types have their interfaces defined in terms of entry and exit ports. Interconnection between components in each component interconnection fragment are defined using Darwin, the Regis module interconnection language (Kramer et al., 1992).

During the generation of the target system, the domain model is tailored using the KBRET tool, based on the optional features selected by the user. Based on these features, the corresponding object interconnection fragments are selected and integrated to create a target system configuration. Assuming that the component types have been developed, the target system configuration can then be instantiated and executed. An early prototype of the tools interfacing the KBSEE with Regis is currently being experimented with in our lab.

7. Conclusions

This paper has described a domain modeling method and prototype Knowledge Based Software Engineering Environment, which has been developed to demonstrate the concepts of reusable software requirements and architectures. The application domain-independent prototype environment supports the development of domain models and the generation of target system specifications. The environment consists of an integrated set of commercial-off-the-shelf software tools and custom-developed software tools.

The KBSEE has been used for modeling several different application domains. In addition to NASA's Payload Operations Control Center and Transportable Payload Operations

Control Center domains, two other application domains, a factory automation domain and a federated banking domain have been modeled. This demonstrates the viability of the domain modeling approach for developing reusable software architectures from which target systems can be generated. It also demonstrates that the environment is indeed domain-independent.

The wide ranging nature of the domain modeling method and the KBSEE was shown when the environment was used to model a software process modeling domain. The Spiral Model, developed by B. Boehm (Boehm, 1988; Boehm and Belz, 1989), encompasses other life cycle models such as the Waterfall Model, the Incremental Development model, and the Evolutionary Prototyping Model. The key characteristics of a given project, referred to as process drivers, are determined during risk analysis. The process drivers are used to tailor the spiral model to generate a project specific process model. A domain model was developed of the spiral model and then the KBSEE was used as a process model generator. The activities of the spiral model were modeled using objects in the domain model and the process drivers were modeled using features (Gomaa and Kerschberg, 1995b).

8. Acknowledgments

The authors gratefully acknowledge the assistance of S. Bailin, R. Dutilly, J.M. Moore, and W. Truszkowski in providing us with domain-specific information on the POCC. The authors also thank the anonymous referees for their comments on an earlier draft of this paper. The domain modeling research and development of the Knowledge-Based Software Engineering Environment was sponsored primarily by NASA Goddard Space Flight Center with additional support from the Virginia Center of Innovative Technology. Portions of the work were sponsored by an ARPA grant, administered by the Office of Naval Research under grant number N00014-92-J-4038. The Software Through Pictures CASE tool was donated to GMU by Interactive Development Environments, Inc.

References

Batory, D. 1989. The genesis database system compiler: A result of domain modeling. In *Proc. Workshop on Domain Modeling for Software Engineering*, OOPSLA'89, New Orleans.

Batory, D. and O'Malley, S. 1992. The design and implementation of hierarchical software with reusable components. *ACM Transactions on Software Engineering Methodology*, 1(4):355–398.

Biggerstaff, T. and Richter, C. 1987. Reusability framework, assessment, and directions. *IEEE Software*.

Blum, B. 1987. The tedium development environment for information systems. *IEEE Software*.

Boehm, B. 1988. A spiral model of software development and enhancement. *IEEE Computer*.

Boehm, B. and Belz, F. 1989. Experiences with the spiral model as a process model generator. In *Proc. 5th International Software Process Workshop*.

Bosch, C., Gomaa, H., and Kerschberg, L. 1995. Design and construction of a software engineering environment: Experiences with Eiffel. *IEEE Readings in Object-Oriented Systems and Applications*, IEEE Computer Society Press.

Gomaa, H., Fairley, R., and Kerschberg, L. 1989. Towards an evolutionary domain life cycle model. In *Proc. Workshop on Domain Modeling for Software Engineering*, OOPSLA, New Orleans.

Gomaa, H. and Kerschberg, L. 1991. An evolutionary domain life cycle model for domain modeling and target system generation. In *Proc. Workshop on Domain Modeling for Software Engineering*, International Conference on Software Engineering, Austin.

Gomaa, H. 1992a. An object-oriented domain analysis and modeling method for software reuse. In *Proc. Hawaii International Conference on System Sciences*, Hawaii.

Gomaa, H., Kerschberg, L., and Sugumaran, V. 1992b. A knowledge-based approach for generating target system specifications from a domain model. In *Proc. IFIP World Computer Congress*, Madrid, Spain.

Gomaa, H., Kerschberg, L., and Sugumaran, V. 1992c. Knowledge-based approach to domain modeling: Application to NASA's payload operations control centers. *Telematics and Informatics*, 9(3/4):281–296.

Gomaa, H. 1993a. A reuse-oriented approach for structuring and configuring distributed applications. *Software Engineering Journal*, pp. 61–71.

Gomaa, H. 1993b. *Software Design Methods for Concurrent and Real-Time Systems*. Addison Wesley.

Gomaa, H. 1994. Configuration of distributed heterogeneous information systems. In *Proc. Second International Workshop on Configurable Distributed Systems*.

Gomaa, H. 1995a. Reusable software requirements and architectures for families of systems. *Journal of Systems and Software*.

Gomaa, H. and Kerschberg, L. 1995b. Domain modeling for software reuse and evolution. In *Proc. IEEE Computer Assisted Software Engineering Workshop* (*CASE 95*), Toronto.

Jackson, M. 1983. *System Development*. Prentice Hall.

Kang, K.C. 1990. Feature-oriented domain analysis. Technical Report No. CMU/SEI-90-TR-21, Software Engineering Institute.

Kramer, J., Magee, J., Sloman, M., and Dulay, N. 1992. Configuring object-based distributed programs in REX. *Software Engineering Journal*.

Lubars, M.D. 1989. Domain analysis for multiple target systems. In *Proc. Workshop on Domain Modeling for Software Engineering*, OOPSLA'89, New Orleans.

Lyndon B. 1989. Artificial intelligence section. *CLIPS Reference Manual*. Johnson Space Center.

Magee, J., Dulay, N., and Kramer, J. 1994. A constructive development environment for parallel and distributed programs. *Second International Workshop on Configurable Distributed Systems*, Pittsburgh, PA.

Meyer, B. 1987. Reusability: The case for object-oriented design. *IEEE Software*.

Mylopoulos, J., Borgida, A., Jarke, M., and Koubarikis, M. 1990. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4):325–362.

Parnas, D. 1979. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*.

Prieto-Diaz, R. and Freeman, P. 1987. Classifying software for reusability. *IEEE Software*.

Pyster, A. 1990. The synthesis process for software development. In *System and Software Requirements Engineering*, R. Thayer and M. Dorfman (Eds.), IEEE Computer Society Press.

Rumbaugh, J. et al. 1991. Object-oriented modeling and design, Prentice Hall.

Shepherd, A. and Kerschberg, L. 1986. Constraint management in expert database systems. *Expert Database Systems: Proceedings from the First International Workshop*, L. Kerschberg (Ed.), The Benjamin/Cummings Publishing Co., Menlo Park, CA.

Shlaer, S. and Mellor, S. 1988. *Object Oriented Systems Analysis*. Prentice Hall.

Szczur, M.R. 1990. A user interface development tool for space science systems. *AIAA/NASA Symposium on Space Information Systems*.

Wegner, P. 1990. Concepts and paradigms of object-oriented programming. *OOPS Messenger (ACM SIGPLAN)*, 1(1).

Yoon, Pil, J., and Kerschberg, L. 1992. A framework for constraint management in object-oriented databases. Paper presented at the *International Conference on Information and Knowledge Management*, Baltimore, MD.

Yourdon, E. 1989. *Modern Structured Analysis*. Prentice Hall.

Automated Software Engineering 3, 309–345 (1996)
© 1996 Kluwer Academic Publishers. Manufactured in The Netherlands.

Enveloping Sophisticated Tools into Process-Centered Environments*

GIUSEPPE VALETTO
GAIL E. KAISER kaiser@cs.columbia.edu
Rank Xerox Research Centre, 6 Chemin de Maupertuis, 38240 Meylan, France,
Columbia University, Department of Computer Science, New York, NY 10027, United States

Abstract. We present a tool integration strategy based on enveloping pre-existing tools without source code modifications or recompilation, and without assuming an extension language, application programming interface, or any other special capabilities on the part of the tool. This Black Box enveloping (or wrapping) idea has existed for a long time, but was previously restricted to relatively simple tools. We describe the design and implementation of, and experimentation with, a new Black Box enveloping facility intended for sophisticated tools—with particular concern for the emerging class of groupware applications.

Keywords: tool integration, workflow, computer-supported cooperative work, computer-aided software engineering

1. Introduction

Process-centered environments (PCEs) and other task-oriented frameworks (see, e.g., the NIST/ECMA reference model (Reference, 1993)) usually support dialogues between external tools and the environment, which serves as a mechanism for integrating the tools according to their roles in the workflow. We identify three categories of integration methods, with respect to their approach to adapting the tools to the environment:

- *White Box*, where a custom tool is developed as part of a particular environment or a pre-existing tool's source code is modified to match a framework's interface. Custom tools may be prohibitively expensive to develop. Changes to pre-existing tools can often be implemented in a straightforward, repetitive manner, but nevertheless the source code must be available—perhaps an insurmountable difficulty when integrating off-the-shelf tools from independent vendors. The White Box approach is followed by several commercial message buses, most based on either the Field broadcast message server (Reiss, 1990) or

*This work was conducted while Mr. Valetto was a graduate student at Columbia University. Prof. Kaiser was supported in part by Advanced Research Project Agency Order B128 monitored by Air Force Rome Lab F30602-94-C-0197, in part by National Science Foundation CCR-9301092, in part by the New York State Science and Technology Foundation Center for Advanced Technology in High Performance Computing and Communications in Healthcare 94013, and in part by grants from AT&T Foundation, Bull HN Information Systems and IBM Canada Ltd. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US or NYS governments, ARPA, Air Force, NSF, NYSSTF, AT&T, Bull, IBM or Xerox.

the Polylith software bus (Purtilo, 1994). PCTE (Thomas, 1989) and similar framework standards probably require more effort in tool adaptation, or a priori adherence to the standard by vendors, but enable a higher scale of integration. The CORBA interoperability standard (Nicol et al., 1993) is not specifically directed to environment frameworks, and seems best suited to tools explicitly organized as servers—which relatively few are at present.

- *Grey Box*, where the source code is not modified but the tool provides its own extension language or application programming interface (API) in which functions can be written to interact with the environment. Relatively few tools, aside from database management systems, provide such convenience (although see (Notkin and Griswold, 1988)). Dynamic linking coupled with replacement of standard libraries (e.g., for I/O) works for some environments, e.g., Provence (Krishnamurthy and Barghouti, 1993), concerned with monitoring simple events such as file system accesses, but it seems unlikely in the general case that arbitrary tools would happen to fit the protocols of a task-oriented framework. In particular, a PCE requires that task prerequisites be fulfilled prior to performing the task, so mechanisms to detect and/or notify that a task has already been completed are inadequate (Popovich, 1992).
- *Black Box*, when only binary executables are available and there is no extension language or API. In this case, the environment must provide a protocol whereby *envelopes* extract objects and/or files from the environment's data repository, present this data to their "wrapped" tools in the appropriate format, and provide the reverse mapping for updated data and tool return values[1]. Envelopes may also be used in conjunction with Grey and White Box methods, but are mandatory for Black Box integration.

Our primary goal in this paper is to augment enveloping concepts and technology to apply to a much wider array of tools. We concentrate on the Black Box model, since it is often the only choice (particularly for legacy tools) as well as the most challenging.

Typical Black Box enveloping technology expects the tool integrator to write a script or program that handles the details of interfacing between the tool and the environment framework, often both to respect the environment's notion of task and to access its data repository, as well as the actual invocation of the tool with an appropriate command line and collection of any outputs and return values. In the case of a PCE, the process definition determines the workflow within which such a script or program may be executed. For example, the task's prerequisites may need to be satisfied in advance and its obligations fulfilled afterwards. The state of the on-going process execution usually sets the context for providing parameters to the tool and determines what should be done with its results.

This approach works well for tools, such as the standard UNIX toolset, that accept all their arguments from the command line at invocation, read and write some files (whose file system pathnames are given on the command line), and return a simple status code. Notice this does *not* preclude interactive tools—even graphical user interface tools such as project schedulers and drawing programs—since the tool's own user interface appears on the user's display device when the envelope executes the tool. The user may then enter text or click menu items as desired; however, the granularity of access to objects/files from the environment's data repository is the entire tool invocation. In other words, the nature of current Black Box enveloping technology requires that the complete set of arguments from the repository is supplied to the tool at its invocation and that any results to be returned to the repository are gathered only when the tool terminates, so that the tool execution—what we call here an *activity*—is encapsulated within an individual task.

There are numerous tools whose natural and/or convenient use doesn't fit this description, but may be highly desirable to integrate into PCEs, including at least the following categories. Note these classes are not disjoint.

- Tools intended to support *incremental* request of parameters and/or return of (partial) results in the middle of their execution, such as multi-buffer text editors and interactive debuggers. Although such tools by definition allow submission of an arbitrary sequence of the user's choice of commands during their execution, when run in a stand-alone fashion, current enveloping technology does not permit the sequence of commands to be guided, automated or enforced by a task-oriented environment, and often even precludes retrieval of their parameters from the environment's data repository (e.g., if the process engine controls all access to the repository).
- *Interpretive* tools that maintain a complex in-memory state reflecting progress through a series of operations: Lisp applications, such as "Knowledge-Based Software Assistant" (KBSA) systems (Chase and Reubenstein, 1992), are classic examples. Such tools may require severe start-up overhead and command substantial system resources (thus we refer to them as "heavy-weight"). We are particularly concerned with permitting *different* users to submit activities to the *same* tool execution instance, even when that tool was not designed to support multiple users. One of our goals is to extend a variety of single-user tools to (modest) multi-user operation.
- *Multi-User* tools, such as conventional database management systems that guarantee atomicity and serializability of separately transmitted but concurrently executing transactions. An important subclass is *Collaborative* tools (often referred to as computer-supported cooperative work—CSCW—or groupware), which abhor the conventional isolation model and directly support multiple users interacting with each other, such as WYSIWIS (what-you-see-is-what-I-see), IBIS decision support, Fagan-style document inspection, desktop video conferencing, etc. (see (Kaplan, 1993; Transcending, 1994) for more examples).

We introduce a *Multi-Tool Protocol* (MTP), where *Multi* refers to submission of *multiple* activities to the same executing tool instance and enabling of *multiple* users to interact with that same tool instance. Tool instances may operate for an arbitrary period of time, far beyond the length of an individual activity on behalf of an individual user; thus we refer to the executing tool instance as "persistent" with respect to the duration of the activities submitted under the MTP protocol. MTP also addresses *multiple* platforms: submitting tool invocations to machines other than were the user is logged in, e.g., when operating over a heterogeneous collection of workstations and server computers but executables are available for only a particular machine architecture or even only for a specific host; and *multiple* tool instances: managing a set of executing instances of a tool, e.g., when licensing limits the number of instances that can operate at the same time (common with commercial server licenses). MTP, as currently defined, treats tools in a Black Box manner. MTP has been implemented as part of the Oz process-centered environment.

Section 2 supplies brief background information on Oz. Section 3 introduces a tool modeling notation for specifying the category and special requirements of the tool; this notation extends Oz's previous facility, but could readily be adapted to other PCEs with some notion of tool declaration. Then we present our main work in Section 4, covering the general ideas, persistent tool sessions for four different categories of tools, an extension of the Oz client/server architecture for managing MTP tools (intended to be adaptable to other client/server or peer-to-peer architectures), the protocol for interaction between a process or task management engine and executing tool instances, and finally the structure of the tool wrappers themselves (we will use the terms "envelopes" and "wrappers" interchangeably throughout the paper). Then Section 5 describes four tool integration experiments, one for each of our categories. We discuss related work in Section 6. The paper concludes by summarizing our contributions and outlining future work.

2. Oz background

Oz (Ben-Shaul and Kaiser, 1994) is a process-centered environment framework. It represents both product (project artifacts) and process (workflow status) data using a home-grown object-oriented database management component, with a separate objectbase for each instantiated process. An object may contain zero or more file attributes, each typed as either `text` (ASCII) or `binary`. The value of a file attribute within an objectbase is a file system pathname into a "hidden" file system specific to that objectbase, not intended to be accessed except through Oz. Non-file attributes include the usual primitive values (strings, integers, etc.), containment of child objects, and references to arbitrary objects elsewhere in the same objectbase.

Oz's Shell Envelope Language (SEL) (Gisi and Kaiser, 1991) is typical of current Black Box enveloping facilities, which typically involve some scripting language[2]. The process engineer (or environment builder) writes what are essentially UNIX *sh*, *csh* or *ksh* scripts, using added constructs that a translator expands into regular shell commands to handle the details of interfacing between the tool and the environment framework. An SEL envelope is associated with each task activity. After parameters have been bound and other preliminaries completed, Oz's process execution service directs that the named envelope be invoked on the arguments specified by the encapsulating task, including literals and/or object attributes. When the envelope terminates, it returns a status code and (optionally) result values to the process engine, at which point the pending task assigns the result values to objectbase attributes and performs various operations based on the envelope's status (typically indicating success versus failure).

The mechanisms described above are implemented within a client/server architecture, one server per instantiated process, as shown in figure 1. Tool envelopes are forked by clients. The server sends envelope names and arguments to the client responsible for that activity, and then handles other clients in a first-come-first-served manner until the tool completes and the results returned by the client arrive at the front of the server's request queue.

The figure shows the main components of an Oz server: Inter-Process Communication (IPC) with its clients, Object Management System (OMS), Software Process Manager (PM), Transaction Manager (TM), and the "glue" that holds them together as well as performing

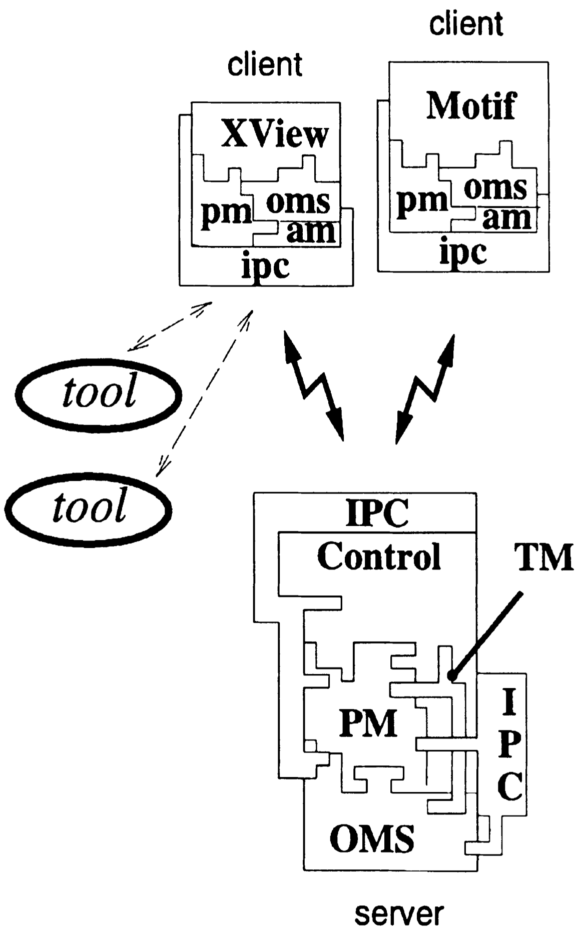


Figure 1. Original Oz architecture.

multi-client scheduling (labelled Control). The clients have limited knowledge of object management (om) and process management (pm), and of course also include an interprocess communication component (ipc); the activity manager (am) is responsible for managing tool invocations. XView and Motif graphical user interfaces are supported, as well as a tty command line interface (not shown in figure). The various components are drawn as "jigsaw pieces" to denote numerous connections among components as opposed to, say, a purely layered architecture. See (Ben-Shaul et al., 1993; Ben-Shaul and Kaiser, 1995) for additional details.

3. Tool modeling

Assuming both SEL-like enveloping and a new MTP protocol are available, the process or other task-oriented execution service needs to specify which tools require which protocol. In

```
<tool-name> :: superclass TOOL;
  [protocol     : (MTP, SEL);
   path         : <string>
   host         : <string>
   architecture: (sun4, ...);
   instances    : <integer>;
   multi-flag   : (UNI_QUEUE, MULTI_QUEUE,
                   UNI_NO_QUEUE,
                   MULTI_NO_QUEUE);
  ]
   <activity-name> : string =
      "<envelope-name> <parameters locks>";
   <activity-name> : string =
      "<envelope-name> <parameters locks>";
   ...
end
```

Figure 2. Modified tool definition notation.

principle, every tool could be invoked via the new MTP protocol, but we retained SEL for Oz (or the equivalent facility for some other system) as the default because we believe that MTP is complementary to SEL on a per-tool basis: together, they address with greater specificity the peculiarities of diverse families of applications, and the choice allows minimization of overhead balanced across a number of factors (see Section 4). In general, we believe an approach to integration based on *multiple enveloping protocols* is likely to achieve the greatest generality.

In the Oz implementation, the tool declaration notation has therefore been modified to include the new portion shown between square brackets ("[· · ·]") in figure 2, which is optional and may be omitted for SEL (some but not all of these fields are meaningful for SEL, as explained later, but defaults are assumed if they are not provided by the process engineer). Note each tool declaration is represented as a subclass of the built-in `TOOL` class; running tool execution instances are viewed as instances of these subclasses (although they are not currently reified in Oz's objectbase).

The new fields have the following meanings:

- `Path`. Indicates the pathname in the file system where the tool's envelope resides (or the tool's own binary executable, since an envelope is not always needed for tool initialization when using our MTP protocol, depending on the details of the tool). For example, an envelope might prompt the user for tool parameters not managed by the environment (such as a database volume).
- `Host`. An Internet address, given when it is necessary to run the tool on a specific host because of some restriction (perhaps due to pragmatic licensing issues).
- `Architecture`. Used to indicate the machine architecture and/or operating system on which the tool (and its corresponding envelope) is expected to run. When the `host` is not specified, the system refers to the `architecture` specification and separate

environment instance-specific configuration information, to determine a corresponding default machine on which the persistent tool (and its envelope) will be invoked.

- `Instances`. This specifies the maximum number of copies of the tool that can execute at the same time (0 means there is no upper limit). Independent of licensing issues, this could be used to bound the system resources allocated to a heavy-weight tool in all its instantiations.
- `Multi-flag`. This determines the behavior of MTP in managing the interactions between multiple human users and a persistent tool instance. We distinguish among four categories of tools, with respect to their single-user versus multi-user and single-tasking versus multi-tasking capabilities, through the cross-product of two orthogonal dimensions:

  - **UNI** versus **MULTI**: MULTI (multi-user) indicates that the same instance of the program can be *shared* by several users, whereas UNI (single-user) allows only for isolated work of each user on his/her own executing instance of the tool;
  - **QUEUE** versus **NO_QUEUE**: where *concurrent* (overlapping) execution of multiple activities with respect to the same tool instance is supported for NO_QUEUE (multi-tasking) but not for QUEUE (single-tasking).

It may seem counterintuitive to think of these dimensions as orthogonal. In the case of `MULTI_QUEUE`, i.e., multi-user and single-tasking, multiple activities on behalf of different users can share the same tool instance, but only one actually controls it and views the user interface at a time, in "floor-passing" fashion. For `UNI_NO_QUEUE`, i.e., single-user and multi-tasking, multiple activities can execute simultaneously in the same tool instance (perhaps in distinct "buffers" or other tool-specific contexts—the tool need not be implemented using multi-threading or parallel processing technology), but all must be on behalf of the same user. The four cross-product cases are explained by relatively generic examples in Section 4.1 and correspond to specific experiments elaborated in Section 5.

Each of the declarations following the brackets specifies the name of a activity together with the file name of an envelope, distinct from the one that started up the tool (if any). The activity-specific envelope is invoked whenever the corresponding activity is submitted to the persistent tool. There are likely to be several qualitatively different activities that can be performed using the same tool, so it is expected that multiple activity/envelope mappings would be listed in the tool declaration. If so, multiple instances of the same activity or several entirely different activities can be submitted to the same persistent tool execution. Formal parameters and locking information are also listed (transaction management is outside the scope of this paper, see (Barghouti, 1992; Heineman and Kaiser, 1995)). The envelope specified by the associated task handles the passing of arguments back and forth to/from the environment's repository as well as the details of interaction with a tool that is already running.

These declarations appear in identical form in SEL specifications, but in that case each envelope invokes a distinct tool instance to perform the activity (and envelopes may be grouped into the same tool declaration for abstraction reasons, without necessarily employing the same external application program). We made no changes at all to Oz's process definition facilities other than the tool declaration notation, and our approach is intended to be orthogonal to the environment framework's mechanisms for workflow definition and performance.

4. The integration protocol

We adopted what we call a *loose wrapping* approach, as opposed to the *tight wrapping* currently effected in Black Box enveloping schemes. The latter relies on complete *encapsulation* of all of the tool's actions inside a single envelope, whereas the former is instead based on *control* of the tool's behavior (from the viewpoint of the PCE), with the enveloping facility intervening only as the need arises during workflow execution and/or upon detection of some external event relevant to the environment. A typical example of the former is when the initiation of a process step (either automatically or through an environment command selected by a user) requires the tool to perform some work, and of the latter when a tool action saves some files that should be recorded in the environment's repository.

Control, as opposed to encapsulation, provides a means for long-lived and intermittent dialogue between external tools and the environment; meanwhile, the tools continue their execution effectively detached from the environment framework. Tight wrapping, on the other hand, governs all phases of a tool's execution, from the moment of invocation to termination; to perform multiple activities using the same tool, it must be explicitly and repeatedly instantiated (even if on behalf of the same user) each time an activity is assigned to the tool.

Our approach may be viewed as combining the advantages of conventional Black Box enveloping and event notification systems like Field and YEAST (Rosenblum and Krishnamurthy, 1991), where tools execute persistently but the server's concern is only for events of interest to other tools and there are no separate "environment commands" or "workflow" that control tools. The Forest extension of Field manages the propagation of event notifications among tools according to "policies" (Garlan and Ilias, 1990), analogous to Oz's process management services, and Provence is implemented on top of MARVEL (Kaiser et al., 1988; Heineman et al., 1992), the predecessor of Oz, but neither has any means for requiring satisfaction of task prerequisites. These systems also do not address one of our foremost requirements, to integrate multi-user tools, and few message buses are concerned with groupware or even support multiple users per bus. Buses internal to PCE frameworks such as ConversationBuilder (Kaplan et al., 1992) and ProcessWEAVER (Fernström, 1993) are exceptions.

Once we established loose wrapping as the overall principle on which to base our design, we analyzed the major capabilities needed to implement our tool modeling facilities (described in the previous section). We divide these functions into two categories: those generally concerned with Black Box integration—i.e., the abilities to invoke and terminate an instance of a tool on demand, to parameterize that instance according to the corresponding process task, to transform objects from/to the environment's representation to/from that required by the tool, to support and display the I/O flow between the wrapped program and its user(s)—and those abilities especially necessary given the nature of the four tool categories of interest (i.e., the cross-product of UNI vs. MULTI and NO_QUEUE vs. QUEUE):

1. Limit the number of co-existing (executing) copies of a given tool according to the specifications set out in the tool's declaration, and to record and service previously unsatisfied requests as soon as possible;

2. Exploit the persistence of MTP-tools, in order to share a given instance among multiple users—possibly emulating partial multi-user capability for programs not usually employed for groupware;
3. Coordinate overlapping requests for access to an instance of a persistent tool from separate users, to avoid deadlocks and starvation on the one hand, and of unintended concurrency of several activities for programs that do not support any form of multitasking on the other; and
4. Record results of intermediate steps of the tool's processing, during the execution of each single activity.

To fulfill these requirements, we have introduced several extensions to Oz's process management services. Analogous extensions could be made to other environment frameworks.

4.1. Tool sessions

To encompass both serial and concurrent access to a tool instance, we introduce *sessions*, which define the life-span of a persistent tool. A session normally begins with an `OPEN-TOOL` command and ends with a `CLOSE-TOOL` command, as illustrated in figure 3. A session's body is made up of a set of activities, denoted `MTP-activity` in the figure, determined dynamically as the users carry out their work within the environment. Note that although the activities are listed in sequence, they could potentially overlap (for NO_QUEUE tools).

`tool` could refer to any tool declared as MTP. The `session` identifier distinguishes among simultaneously executing instances of the same persistent tool, so that multiple users can choose to participate in a particular session opened by another user (for MULTI tools). Both arguments are selected from menus. Users can ask to join an existing session (if there are any) by clicking the corresponding automatically generated session identifier when issuing an `OPEN-TOOL` command, or request a new session as shown in figure 4. The current implementation does not provide any support for access control, e.g., specifying which users are permitted to, or are required to, join a particular session. There is also no support for providing parameters for tool initialization from *within* the environment, which is less limiting than it sounds since the process steps that trigger incremental interaction with the tool usually provide arguments from the environment's repository.

Leaving a session is achieved with a `CLOSE-TOOL` command applied to a session where there are still other active users. In this case, the `CLOSE-TOOL` does not kill the tool instance, but only changes internal information about the association between the user and the session. Termination of the program follows the `CLOSE-TOOL` command of the last participant.

```
OPEN-TOOL tool [session]>
    <MTP-activityA> <argumentsA> <session>
    <MTP-activityB> <argumentsB> <session>
    ...
CLOSE-TOOL <tool [session]>
```

Figure 3. Tool session template.

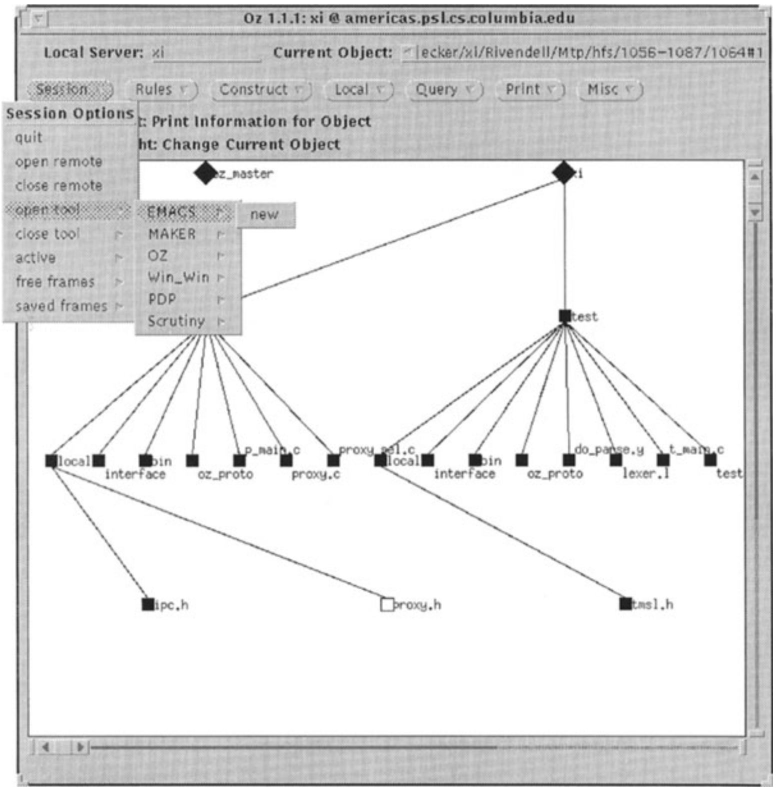


Figure 4. Oz MTP interface.

Besides setting the duration of a specific tool instance and providing a context for sharing an application, sessions are central in several other functions supported by our MTP protocol. For example, they implicitly operate on what we call the *Session Queue* of a tool. This feature allows us to satisfy the constraints posed by the `instances` field of a tool declaration, accordingly limiting the maximum number of copies of the program that can be active simultaneously. (Such a restriction could be violated due to tool instances executing completely outside the environment, resulting in tool invocation failures.) When `OPEN-TOOL` is issued, the system first checks whether the request is satisfiable given this constraint. If the limit has been hit, the request is not serviced, but is recorded in the Session Queue; when an already running session is terminated, the next queue entry is extracted and automatically initiated (the user is effectively notified when the user interface of the tool pops up on his/her workstation monitor).

Our design also allows for a special case where it is possible to use a persistent tool without being compelled to issue the `OPEN-TOOL` and `CLOSE-TOOL` commands every time, via an implicit *atomic* session that consists of only a single activity. *Atomic* sessions are instituted by the system, transparently to the user, when a user intends to perform an activity associated with an MTP tool but has not previously opened or joined a session. In that case, an implicit OPEN-TOOL command is automatically executed and the new tool instance is marked as *atomic* by the environment, so that no other activities (or OPEN-TOOL/CLOSE-TOOL commands) can be directed to it. When the activity finishes, the tool is shut down automatically.

Our sessions idea leads to a number of questions on how different users could, practically, participate in the same session of a persistent tool, thus exploiting the same resources and the collected state of the executing tool. In our MTP design, we stressed the facets intended to accommodate in a natural way those applications that are inherently designed for collaboration, or—a more ambitious goal—to exploit in a multi-user context those tools that, even if not commonly employed in that manner, the environment builder considers adaptable to and promising for collaborative activities.

Our four categories of tools provide a flexible solution to these problems: the valid values of the multi-flag field within the tool modeling specifications represent and enforce in the protocol four working models, intended to cover as widely and as precisely as possible the behaviors and requirements of various classes of persistent tools.

UNI_QUEUE is the most basic category: with it, we intend to accommodate applications that are strictly single-user and that could not adequately support concurrent operations deriving from simultaneous MTP activities. Therefore each instance of such a tool is reserved exclusively to the user who requested it in the first place, via an OPEN-TOOL command, and the body of the session is made up of a simple sequence of activities that are never permitted to overlap.

The most significant difference between MTP's UNI_QUEUE and SEL is that multiple operations can be sent to the same copy of the tool, under the control of the process engine, by exploiting the newly introduced concept of *Activity Queues*: each UNI_QUEUE session is associated with an Activity Queue, which holds in first-come-first-served order the activities waiting to take control of the tool instance.

Consider, for example, a drawing program with a relatively long start-up time (e.g., it may load numerous fonts during initialization). Rather than force the user to wait several seconds to bring up the tool for each of the increasingly detailed data flow diagrams the process directs him/her to construct as part of a design document, the tool is invoked once and then this executing instance is reused for each separate diagram. This model assumes the tool provides interactive commands to load and store particular diagrams in the file system or a database, as most drawing programs do. Each activity begins by loading an existing diagram, indicating that a clean slate is needed, or simply expanding on the most recently loaded diagram, and ends with storing that diagram, with arbitrary tool-specific commands in between.

UNI_NO_QUEUE is intended to satisfy more complex integration requirements and to allow for more operational flexibility. Again, each tool instance is reserved for just one user, but the full exploitation of the inherent multi-tasking (or multi-context) capabilities of the tool is supported, by directing to the tool multiple simultaneous or overlapping activities.

One case is a multi-buffer text editor, where the user can easily switch among buffers with an interactive command; perhaps two or more buffers can be shown at the same time. A programmer might be part way through editing a particular source file when he /she realizes that it would be useful to cut and paste some code from another file, and modify the copied code, rather than type it in from scratch or call that other code as a subroutine. And while looking at this other source file, the programmer decides to make some changes to it, too, which may entail loading into the editor the header file(s) it imports, and so on. The process dictates certain obligations, such as recompilation, static analysis, and/or code inspection, for each edited file, perhaps somewhat different process segments depending on file type (source vs. header vs. documentation) and/or on whether the programmer is the "owner" of that file. Thus the editing of each file must be treated as a separate activity by the process, while at the same time it is useful to load the files into different buffers of a single executing instance of the editor rather than bring up a separate instance for each file.

If a tool is not inherently multi-user (as is the case for most current tools), but is declared `MULTI_QUEUE`, only the most rudimentary form of sharing is possible: different users are allowed to join the same session, and therefore to access the same executing tool instance. But they must "take turns" (if they happen to issue requests that overlap in time): they are forced to wait in the Activity Queue until the previous activity is finished. Note that users whose requests are placed in the Activity Queue may still execute other process steps—or decide to abort and try again later (Oz's XView and Motif interfaces allow a user client to context-switch at will among in-progress process segments, and many other environments do likewise). Albeit limited, this form of sharing can be usefully exploited in various collaboration scenarios, for example, by multiple users committed to take care of different sequential stages of the same complex, long and composite process task, in which all must employ the same external program. One can then think of the `MULTI_QUEUE` tool as a semi-permanent environment service for these users.

Any interactive tool could, in principle, be supported by MTP as a `MULTI_QUEUE` tool. But it would not always be particularly desirable or useful to do so. Imagine declaring an electronic mail tool as `MULTI_QUEUE`. Then one user might read and respond to one incoming message, another user the next message, a third composes a new outgoing message, and so on. But such an activity sequence seems unlikely to be part of any practical software development process. Instead, `MULTI_QUEUE` is intended primarily for tools that build up a substantial in-memory state and that—under normal usage—support a sequence of activities that depend, at least in part, on the state constructed by previous activities and on the efforts of distinct human users (or user roles).

One example might be a Lisp-based application that generates natural language, say for a user manual, from a knowledge representation constructed during the requirements analysis and functional specification phases of the software process. A sequence of human-directed procedures are generally needed to turn the internal structure into prose appropriate for the end-user of the system under development. A software analyst might initiate the work, perhaps interleaved with activities performed by programming and/or quality assurance personnel, to be polished off by a technical writer and reviewed by a customer representative. Each user begins his/her activity where the last left off, with the tool's user interface automatically redirected among user display devices as another user takes over. The different user roles bring different kinds and levels of expertise to bear on producing the finished document. Note that while it is certainly possible to develop a knowledge-based assistant that saves its relevant state in the file system between steps, allowing separate invocations for each user, a given tool is not necessarily constructed that way. Further, even if such were available as an option (e.g., a Lisp image might be saved on disk), the heavy-weight start-up overhead might be best limited to a single invocation per process segment rather than once for each activity.

The `MULTI_NO_QUEUE` class was conceived to accommodate inherently multi-user systems, taking into account their architectural and functional peculiarities. MTP ensures in this case that every `OPEN-TOOL` command issued by some user in the context of the same session maps to the instantiation of a portion of the same multi-user system (e.g., a client in a client /server architecture), which is assigned to that user.

While MTP is in charge of directing users' process-determined activities to `MULTI_NO_QUEUE` tools, it is the intrinsic multi-user nature of these applications that defines whatever sharing and concurrency control policies are necessary to operate in the multi-user and possibly collaborative context. The transparency or visibility among user-controlled components with respect to their activities and data depends solely on the nature and the purpose of the tool, which may support collaboration (in a groupware application) or enforce isolation (in a conventional database management system). The integration protocol, per se, is not concerned with these issues.

An interesting `MULTI_NO_QUEUE` case is a process-centered environment, itself treated as a tool. The controlling PCE might specify the process at a relatively coarse granularity, e.g., coding and unit testing an individual module would be represented as a single task and integration testing of a subsystem as another. The controlled PCE (i.e., the "tool") might assist the users in carrying out the finer details of such tasks, e.g., editing, compiling, constructing test harnesses, and debugging would be separate steps triggered by the code-and-test task. (We have explored elsewhere the advantages of integrating higher level and lower level process definitions (Kaiser et al., 1994).) We assume here that the controlled PCE is itself designed and implemented as a multi-user system, e.g., following a client/server architecture as in Oz, to allow teamwork within each coarse-grained step as determined by the finer-grained process. The two PCEs may or may not be distinct instances of the same system.

4.2. Architecture

The implementation architecture is necessarily specific to Oz, but we anticipate that a similar approach would apply to other multi-user process-centered environments. We divided Oz's clients into two categories, new *proxy clients* (or just proxies) and the original *user clients*.[3] Proxy clients introduce into the architecture a new kind of long-lived entity, with the role of spawning, managing, and achieving the integration of persistent tools. User clients are always associated with human users of the system, who invoke and exit them at will, and therefore they cannot be relied on to support the life cycle of a persistent tool instance. The Oz server persists indefinitely but provides process execution and object management services and most aspects of tool management discussed in this paper, but is intentionally not directly involved with tool invocation (in part for performance reasons, see (Ben-Shaul, 1991)).

In our design, the session management commands (`OPEN-TOOL` and `CLOSE-TOOL`) are issued by user clients on demand by human users and executed by the appropriate proxy client, installed on the machine determined by the `host` or `architecture` data in the MTP `TOOL` declaration and, if both fields are null, then on the same machine where the Oz server is running. Subsequent activities submitted to the same tool may be initiated from a user client's user interface, but are delegated to the proxy client. The same proxy manages all persistent tools executing on the same host (with respect to activities managed by the same Oz server).

Proxy clients do not need to interact directly with any human operator, so no user interface is needed. However, they must manage the user I/O to/from persistent tools. This involves redirection of simple textual I/O between the tool and the user client, and more significantly the ability to display the tool's own graphical user interface (GUI) on the user's display. Most inherently multi-user tools are able to dispatch private instances of their user interface to each user, but for other tools (e.g., originally single-user tools extended by MTP to a modest form of groupware) we exploited the public-domain *xmove* utility (Solomita et al., 1994), which transfers the GUI of a tool across workstations and X terminals. Resetting the X Windows *DISPLAY* variable would be insufficient, since the GUI instance has to start on one display device for one user, then move to another for a second user, etc. *without* reinitializing the tool. (Note our implementation is inherently limited to those GUIs based on X Windows.)

Another job assigned to proxies is to spawn, manage, and communicate with auxiliary programs called *watchers*, each of which operates in the temporary directory for a tool instance and "notices" any files created or updated by a tool. These files are mapped to activity arguments according to a configuration file constructed by the envelope. The files can then be transferred back to the environment when the activity is completed.

The new proxy client, here supporting `MULTI_QUEUE` operation for a single persistent tool, is depicted in figure 5. The internal composition of a proxy client is nearly the same as a user client, except there is no user interface and an additional component handles watchers, activity queues and other aspects of persistent tool management (the unlabelled piece of the proxy client in the figure). The same proxy client may manage multiple persistent tools, in which case there may be multiple activity queues—one for each `UNI_QUEUE` or `MULTI_QUEUE` tool.

Besides the capability for the same tool instance to handle multiple activities, another major difference between a SEL-like protocol and MTP's UNI cases, at least with respect to environment frameworks similar to the Oz architecture, is forking of the envelope and, indirectly, the tool by a proxy client—often not on the same machine as the user client—which could result in unnecessary communications overhead. MTP could easily be modified to default to a proxy on the same machine as the user client, and even some of the user and proxy client functions could be merged so that a separate proxy would not be needed when `host` and `architecture` specifications are not supplied and/or match the user client machine.

4.3. Envelope execution

The most significant remaining issue that must be resolved to complete the design of our new protocol is the way in which the execution of envelopes is accomplished, in the manner of the *loose wrapping* concept. A typical MTP activity execution steps through the sequential

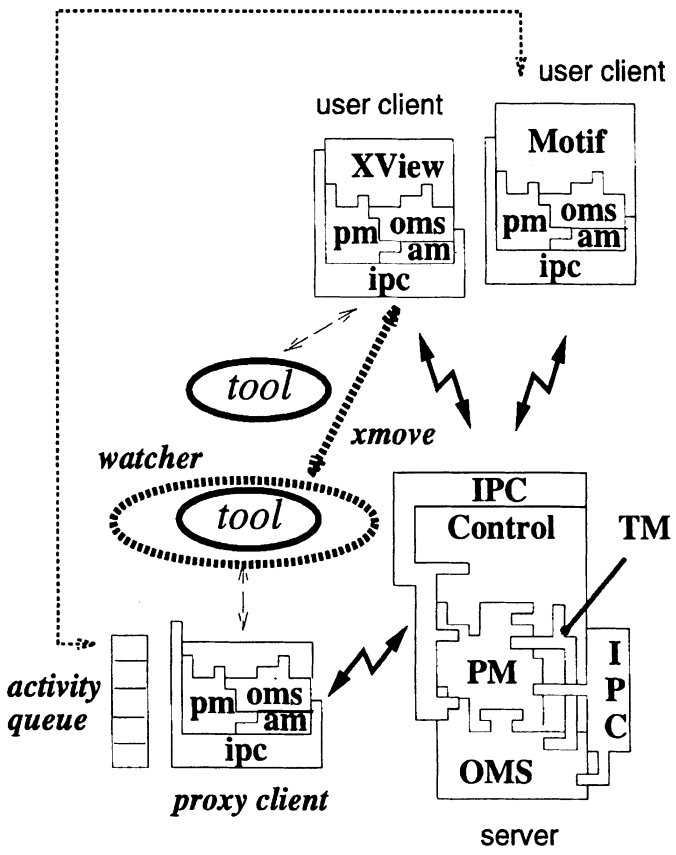


Figure 5. New Oz architecture.

phases listed below:

1. A **reservation** phase, in which a tool session is acquired on behalf of the activity and its associated user. This is carried out according to the session mechanism explained above.
2. An **initialization** phase, in which the objects/files from the environment are made available to the tool and any other parameterization functions are performed. We have employed for this purpose a standard envelope template, which accepts as its parameters: pathnames corresponding to file attributes in Oz's object management system; the path to a dedicated temporary directory that is created when the tool is started up and within which it normally operates; and some additional information used for internal housekeeping. The filename of this envelope is given by the tool declaration in its `envelope-name` field.

   The envelope is forked by the relevant proxy client, which sets up UNIX pipes for communication. The first job of the shell script is to copy the files into the tool's dedicated directory, thus making them visible to the tool; then any series of shell commands can be inserted, to perform whatever customization is necessary; finally, via the pipes, a sequence of text messages is sent to the proxy, to be displayed to the user in a pop-up window. These messages may include the values of primitive attributes from Oz's objectbase, and are intended to direct the loading of the files from the temporary directory into the memory of the application and otherwise instruct the user as to what to do. For example, the text presented in the window might indicate the command line or the mouse action that the user should enter to get started on the activity, although the details of performing the work are usually left to the user's own creativity and expertise.

Although we would have preferred a totally automatic loading procedure, as accomplished by SEL, that it is hardly possible given the inherent restrictions of the Black Box model: MTP tools are already running before the execution of any activity envelope, and therefore cannot be initialized according to the individual activities. Moreover, we cannot assume any special facilities on the part of the tool for simulating user input; redirecting "stdin" is generally insufficient for GUI tools. However, the envelope, via messages to the pop-up window, may still provide assistance and guidance to the users in a practical and convenient manner.

A Grey Box variant of MTP would overcome this drawback, since the tool's programmable facilities could act in collaboration with the envelope, producing and exchanging messages interpreted as directives to be executed by the tool. (Some Grey Box experiments have been conducted using SEL; see Section 5.2.) In the White Box case, this issue can usually be avoided entirely.

3. An **operation** phase, which includes free use of the tool with all its features, including manipulation of the loaded data. There is no restriction on the use of the tool, because it is accessed directly and not through any intermediary. The only requirement of the MTP protocol (that cannot, however, be enforced in the Black Box case) is that the execution must not be terminated through the tool's own internal command, menu button or procedure, but only via the environment's `CLOSE-TOOL` command. In addition, both MTP and SEL assume that users do not access the "hidden" file system sereptitiously, e.g., loading files into the tool outside the workflow, although there is nothing beyond an obscure organization and naming scheme (witness the filename the user is asked to type in figure 9) to prevent them from doing so.
4. One or more **data recording** phases may interleave with other actions, whenever the user saves temporary results of the work he/she is performing (the tool updates the copies of the files kept in its own temporary directory, and not those internal to the environment). Such events are monitored by the proxy client's watchers. A table of updated files is maintained in the proxy and used in the next phase.
5. The **conclusion** of the activity, at which point control of the tool is released (with respect to this activity). The user is required to designate the activity's completion as either a `success` or a `failure`, via corresponding buttons in an MTP-specific extension to Oz's activity management window (see figure 9). The data resulting from the execution is stored back in the environment only if the user considers the activity successfully completed.

SEL expects the envelope to automatically capture the return code of the tool after the user decides to close it, but in MTP the tool remains indefinitely active; therefore the only means of ending an individual activity is to let the user decide when his / her work is finished and to provide a way to communicate this fact (and how to handle the results) to the envelope. Other differences are that SEL file updates are permanent, regardless of the success or failure status: actually, SEL may return any value in a range determined by the encapsulating task, each of which will result in different obligations following that task. A similar facility could be added to MTP.

4.4. *Wrapper structure*

Envelopes provide a very flexible approach to tool integration. Consisting of either standard scripts in some scripting language (as we have employed for MTP), or augmented variants of the scripting languages that provide primitives to handle interfacing to the environment and its data repository (as in SEL)—or possibly even written in a conventional programming language, wrappers offer programmable facilities that can handle the different needs and idiosyncratic properties of a wide range of external applications in a convenient and uniform way.

MTP uses two kinds of envelopes: the first is executed in response to the `OPEN-TOOL` command, whereas the second operates at the granularity of the individual activity. The latter is concerned mainly with preparing and loading the data that must be processed by the program during the associated activity; the former is used to perform customization of the tool, in order to present it to the user(s) in the correct state, in relation to the characteristics of the system and the work model indicated by the `multi-flag` specification. This kind of customization script is usually very simple—no more than a few lines of straightforward shell commands—but sometimes may be quite complicated, accounting for complex interactions with the environment through watchers, and sometimes even for the invocation of other auxiliary (usually simpler) scripts that perform supplementary bookkeeping or actions in response to particular states of the application. An example of an intermediate case is shown in figures 6 and 7; note the latter shows the contents of the auxiliary `close_oz_script` invoked by the former.

In the case of the Oz implementation, the envelope writer must be a relatively skilled shell programmer with some knowledge of the purpose and the functions of the wrapping protocol to be able to easily set up the scripts. The burden might be lowered somewhat if MTP were to extend the scripting language with special-purpose primitives, perhaps somewhat different sets to accommodate each of the four work models. However, the experience gained with SEL shows that even with such primitives the scripts are not exactly trivial, since the intrinsic specificity of the application programs necessitates ad hoc treatment for each case.

Language extension would be useful mainly to abstract and parameterize those operations that must be carried out in a repetitive manner for any application; this seems more plausible with the data interface between the tool and the environment, rather than with the adaptation of their reciprocal behavior. Consider the example shown in figure 8: some of the shell commands, those marked with the comment `# always`, must always be present in any MTP activity-related envelope; others, indicated by the comments that contain the words `FILE parameters`, are needed to handle certain types of incoming data, and are similar but not identical in all the envelopes. These two sets of commands together contribute to preparing the data involved in the activity.

```
#!/bin/sh
#initialize variables
SERVER_PID=-1
CLIENT_PID=-1
# look if already hooked to the environment directory
FOUND='find . -name linkfile -print'
# if environment directory is not found
if ["x$FOUND" = "x"] #no oz_server active
then
      #The OZ environment directory is not set up
      #The shell script exit with -1
      echo "The OZ environment directory is not set up properly"\
          >> /tmp/SPC.log
      exit -1;
else
      #Change to the OZ environment directory
      cd linkfile
      # test whether there is a server running
      SERVER_PORT='find . -name .server_port -print'
      if ["X$SERVER_PORT" = "X"]    #No server is running
      then
            #bring up the oz server
            /u/bleecker/xi/bin/oz_server &
            SERVER_PID=$!
            #Record the server process id
            echo $SERVER_PID>.server_pid
            #Record the number of client run on the server
            echo "0">.client
            sleep 5
      fi
      #start up the client
      /u/bleecker/xi/bin/gpc -x
      CLIENT_PID=$!
      #increase the number of clients
      read CLIENT_NUMBER <.client
      CLIENT_NUMBER='expr $CLIENT_NUMBER + 1'
      echo $CLIENT_NUMBER >.client
fi
CURR_DIR='pwd'
# trap a request to kill this OZ component and
# invoke close_oz_script to take care of this task.
trap '/u/bleecker/xi/Rivendell/Mtp/mtp/close_oz_script \
      $CURR_DIR $CLIENT_PID; exit 1' 2
wait
```

Figure 6. Example initialization script for a multi-user client/server tool.

```
#!/bin/sh
# $1 tool_directory
# $2 oz client process id
echo "Close the client and server!\n"> &2
read CLIENT_NUMBER < .client
if [$CLIENT_NUMBER ! = 1]
then
       CLIENT_NUMBER='expr $CLIENT_NUMBER - 1'
       echo $CLIENT_NUMBER > .client
       kill -9 $2
else
       read SERVER_PID < $1/.server_pid
       kill -9 $2
       kill -2 $SERVER_PID
       #take care of the garbage
       rm $1/.client
       rm $1/.server_pid
fi
```

Figure 7. Example termination script for a multi-user client/server tool.

The other shell commands, marked by the `# tool-dependent` comments, are concerned with operating the tool towards the goal of the task at hand. It is clear that in the general case the size and the complexity of this last set is dependent on the wrapped application, of the supported work model and, especially if a lot of direct interaction with the user is necessary, of the activity to be performed. In contrast, the former two sets are relatively independent of all these factors; hence it would be easier to invent scripting-language extension facilities to express them.

However, it would also be possible (and desirable) to define some ad hoc constructs for use in those tool-dependent statements that communicate to the user the actions that he/she should perform, e.g., to carry out the loading of activity arguments into the tool instance, during the initialization portion of an MTP activity. In figure 8 these messages are implemented simply with `echo` commands prefixed by a common string (`# *** #`); the output is redirected through pipes maintained between the envelope and the proxy client that initiated it, and the proxy is in charge of displaying the messages to the user in a pop-up window. One could certainly imagine more sophisticated facilities for guiding the user.

5. Tool integration examples

To test the facilities described in the previous sections, we have used several available in-house applications and off-the-shelf tools. The purpose of these tests was to gain confidence in the viability of the new MTP protocol, and in particular to challenge its ability to accommodate a wide range of variability in the nature of the wrapped applications.

```
#!/bin/ksh
#input parameters:
# $1 tool dir.          <----- MTP additional parameter
# $2 C file             <----- NOTE: FILE parameter
# $3 compile status     <----- Literal
# $4 compile log file   <----- NOTE: FILE parameter
# $5 C file proto       <----- For later extension to match
# $6 local project tag  <----- SEL editor envelope functionality
# $7 EnDoFAtTrSEt       <----- marks end of arguments from process
# $8 task identifier    <----- MTP additional parameter
# $9 client identifier  <----- MTP additional parameter

LOGFILE="/tmp/ForkLog"                   # debugging code
echo "start up enveloper" >>$LOGFILE     # debugging code
cp $2 $4 $1                    # copy all FILE parameters into the tool dir.
CFile='basename $2'            # for all FILE parameters
CompileFile='basename $4'                # for all FILE parameters
CPath='echo $1/$CFile'                   # for all FILE parameters
CompilePath='echo $1/$CompileFile'       # for all FILE parameters
F_LIST_DUMMY=$1/filelist_tmp                     # always
F_LIST=$1/filetable                              # always
touch $F_LIST_DUMMY                              # always
echo $9 $8 $CFile $2 >> $F_LIST_DUMMY    # for all FILE parameters
echo $9 $8 $CompileFile $4 >> $F_LIST_DUMMY
                                         # for all FILE parameters
echo $F_LIST_DUMMY >>$LOGFILE            # debugging code
FOUND='find $1 -name filetable -print'           # always
if ["x$FOUND" = "x"]                             # always
then                                             # always
      mv $F_LIST_DUMMY $F_LIST                   # always
else                                             # always
      F_LIST_CAT=$1/merge_list                   # always
      cat $F_LIST_DUMMY $F_LIST > $F_LIST_CAT    # always
      rm $F_LIST_DUMMY                           # always
      mv $F_LIST_CAT $F_LIST                     # always
fi                                               # always
echo \#***\#TYPE: CTRL-xf $CPath
                               # tool-dependent : load code file
if [$3 = "NotCompiled"]        # tool-dependent
then                           # tool-dependent
       echo \#***\#TYPE: CTRL-x 2
                               # tool-dependent : display new buffer
       echo \#***\#TYPE: CTRL-xf $CompilePath
                               # tool-dependent : load compiler logfile
fi                             # tool-dependent
```

Figure 8. Example activity script for a multi-tasking tool.

Therefore, we have tried to define the degree of integration that can be reached and to identify limitations (either based on the characteristics of the tool category under examination, or specifically to the adequacy of our support to the single cases) or unresolved problems we need to address during future development. The applications we used as examples were:

- `idraw` as a `UNI_QUEUE` tool, where activities are queued for one-at-a-time execution (the same user may submit activities from multiple Oz clients, and the user interface is transferred among workstation monitors as needed);
- `emacs` as a `UNI_NO_QUEUE` tool where steps are not queued but may overlap (typically on a single monitor);
- A Lisp-based natural language processing system called FUF as a `MULTI_QUEUE` tool, where steps are queued for one-at-at-time execution (and the user interface is transferred among users participating in the same session as needed); and
- Oz itself as a `MULTI_NO_QUEUE` tool (that supplies its own clients for multiple users).

5.1. UNI_QUEUE: idraw

idraw (Vlissides and Linton, 1990) is a popular public-domain drawing tool, commonly used to develop pictures and diagrams stored in a postscript form. It provides an intuitive graphical user interface employing a well-known paradigm based on mouse movement and menu selection to operate on a virtual canvas shown within an X window. *idraw* is intended to be single-user; although it supports multiple buffers, we ignore that feature here, and treat the system as if it were necessary to save the current document before loading a different one. This limited use of *idraw* serves as an example of the category of programs where such restrictions are inherent. From our point of view, *idraw* presents some additional features of interest since it fulfills our definition of heavy-weight tool: there is a relatively long initialization time following its invocation[4].

In our experiment, we employed a distinct activity, parameterized by a file attribute from Oz's objectbase, to construct a complete diagram or to allowing editing of an existing diagram stored in that file, with the details of the drawing left to the creativity and expertise of the user. That is, a activity's envelope sends a message to be displayed in a pop-up window, telling the user to load a file with a particular pathname, and briefly instructs the user regarding the purpose of the drawing to be constructed for that file. The user is responsible for using *idraw*'s normal command to later save that file, prior to announcing the conclusion of the activity. This accounts for a simple interaction model that is common practice in the use of such kind of tools; however, it would alternatively be plausible to invent activities and corresponding envelopes to operate at a much finer level of granularity, for example, "select the line icon and insert a vertical line two inches to the left of the triangle", but we doubt this would be useful (except perhaps as part of a tutorial in the use of a system devoted to the management of graphic documents).

The construction of the corresponding wrapper, and of wrappers for most `UNI_QUEUE` applications, is actually very simple: the only tool-dependent statements are aimed at instructing the user on how to load the input file and (optionally) on what he/she must do with it.

A few words are in order regarding our intentionally restrictive use of *idraw*: we had some trouble finding a good candidate for the most basic UNI_QUEUE category, among the *interactive* tools we had on hand for testing (SEL seems adequate and completely satisfactory for *non-interactive* tools, such as compilers, that must be restarted for each new set of arguments anyway); *idraw* on the other hand seemed to have many of the properties that we were looking for in a UNI_QUEUE candidate. However, we recognize that it would normally be deemed UNI_NO_QUEUE, because of its intrinsic multi-buffering capability (see Section 5.2). Further, one could imagine employing *idraw* in a multi-user context, where one user starts a picture and others add to and finish it, analogous to the work model in Section 5.3, in which case *idraw* could even be designated MULTI_QUEUE.

Given all of the above, one may have the impression that perhaps the UNI_QUEUE category is not really necessary. However, we expect that environment builders will discover cases where they intend a tool to be used in a certain restricted way within the workflow, and enforcement of UNI_QUEUE would prove useful.

In general, UNI_QUEUE appears suitable to deal with those applications that do not present any multi-tasking capability and do not seem particularly adaptable to multiple users, but are most conveniently handled as persistent tools. The main advantages of persistence for this class of tools, and the most valuable improvements introduced by MTP's loose wrapping compared to tight wrapping as in SEL, is the reduction of start-up overhead (since the tool need be invoked only once) and the user can run ordered sequences of activities on the same instance of the program without losing its internal state.

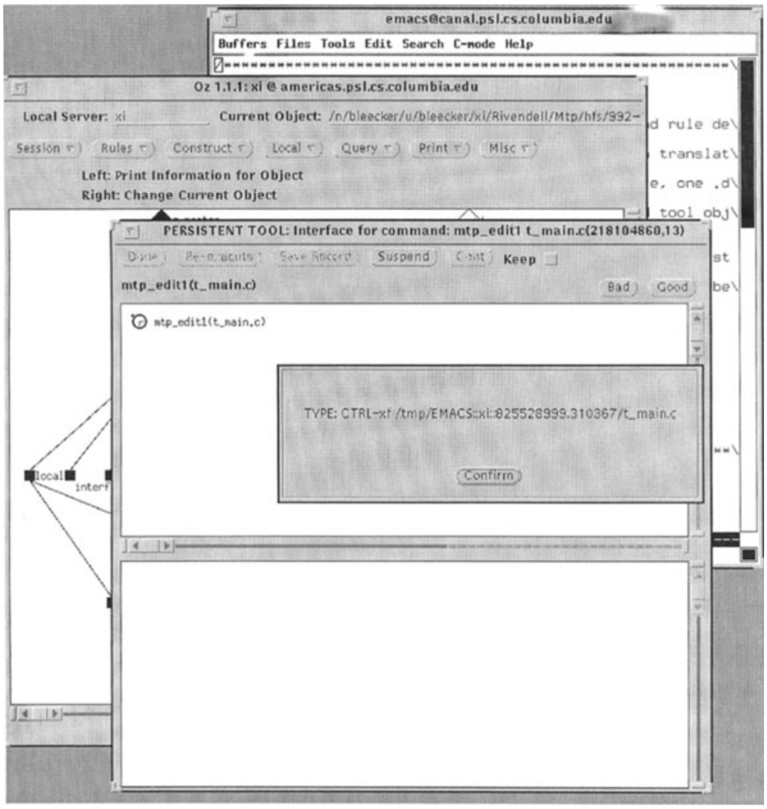
5.2. UNI_NO_QUEUE: emacs

emacs (Stallman, 1981) is one of the most readily available and widely used text editors; its sophisticated functionality and features make it a very useful tool, which nearly reaches in itself the status of a single-user programming environment. All of its commands are expressed with sequences of keystrokes, augmented with mouse pointing and selection; its latest versions also support menu selection, at least for its main features. One of the most useful properties of *emacs*, and one of the most important for us with respect to this discussion, is its buffering capability. This enables the user to operate simultaneously on multiple files, keeping several buffers in the background and switching among them on command. Coupled with the ability to split the display and hence show more than one of the buffers, this feature is of great use to perform complex and incremental editing sessions that involve as many different data sets as needed.

Many users would prefer to use *emacs* in the natural fashion available outside a process-centered or otherwise task-oriented environment framework, which is to create and kill buffers, load and save files, and cut and paste among buffers/files, as the urge arises during perhaps very long work sessions[5]. *emacs* demonstrates the most obvious limitation of conventional Black Box wrappers—that is, all arguments must be supplied on the command line at tool start-up—in which some peculiarities of the application do not fit well with the protocol's design and are left unsupported, but it is nevertheless possible to integrate the program in some form.

MTP's UNI_NO_QUEUE class allows for overlapping multiple activities that involve loading various buffers of the same executing *emacs* instance with the desired files for the user's editing sessions. MTP then employs watchers to allow mapping of each modified file to the corresponding activity and hence discriminates what file attributes must accordingly be modified inside the environment at the end of the activity. The use of a pop-up window during the initialization phase of each activity, and extensions to the standard activity window to indicate completion, effectively isolates the overlapping activities, in the sense that their data flow and status with respect to the on-going process are independent.

In our experiment, we employed individual activities, parameterized by file attributes, to edit programming language or documentation files; the details of the programming or writing were the concern of the user. That is, an activity's envelope would display a message on a pop-up window telling the user to load the file with a given pathname, as shown in figure 9, and perhaps briefly explain to the user the purpose of the code or prose in that file (not shown in the figure). Rather than simply asking the user to edit, the envelope might instead request the user to repair the syntax errors found during the last compilation—by sending a file containing those error messages to another buffer as part of the same activity.


emacs@canal.psl.cs.columbia.edu

Oz 1.1.1: xi @ americas.psl.cs.columbia.edu

Local Server: xi   Current Object: /n/bleecker/u/bleecker/xi/Rivendell/Mtp/hfs/992-

Left: Print Information for Object
Right: Change Current Object

PERSISTENT TOOL: Interface for command: mtp_edit1 t_main.c(218104860,13)

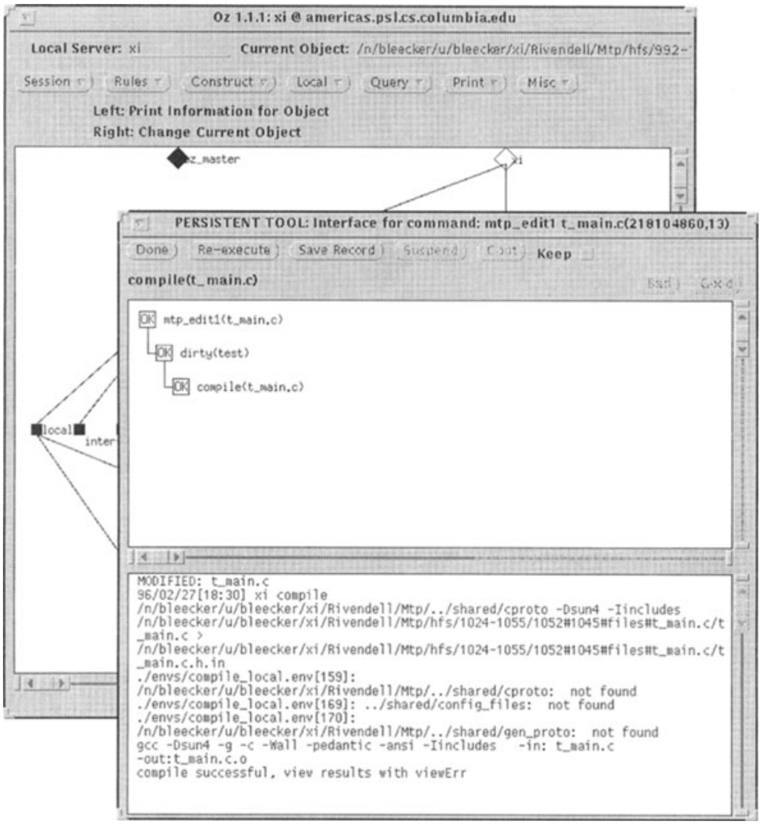
mtp_edit1(t_main.c)

TYPE: CTRL-xf /tmp/EMACS::xi::825528999.310367/t_main.c

Figure 9. MTP activity initiation.

The complete script of an *emacs* wrapper of this kind is shown in figure 8; it performs the loading of a C source file together with the results of the last compiler run, if unsuccessful, to display the generated error messages. Again, the user must give *emacs*'s normal command to save the source file. He/she may choose to indicate that the completion of the activity has been successful, by committing changes to the environment's repository via the **Good** (success) button in Oz's activity window. Then the workflow may automatically continue to other tasks, as illustrated in figure 10, where MTP and SEL activities may be arbitrarily intermingled in a single process fragment. Or the user decides not to save his/her work, by selecting the **Bad** button (failure), which has the effect of withdrawing whatever intermediate saves were performed during the work and noticed by the watchers. As with *idraw*, we did not consider finer-grained activities such as "add a new floating point variable to function *f* and initialize it to *pi*", but the implementation supports them.

A previous attempt to extend Oz's enveloping mechanism had focused on *emacs* as a test case, and tried to resolve the problems posed by the desired incremental data exchange with the environment. This previous attempt exploited a facility not provided by most tools: an


Oz 1.1.1: xi @ americas.psl.cs.columbia.edu
Local Server: xi
Current Object: /n/bleecker/u/bleecker/xi/Rivendell/Mtp/hfs/992-
Session
Rules
Construct
Local
Query
Print
Misc
Left: Print Information for Object
Right: Change Current Object
z_master
xi
local
inter
PERSISTENT TOOL: Interface for command: mtp_edit1 t_main.c(218104860,13)
Done
Re-execute
Save Record
Suspend
Quit
Keep
compile(t_main.c)
Bad
Good
OK mtp_edit1(t_main.c)
OK dirty(test)
OK compile(t_main.c)

```
MODIFIED: t_main.c
96/02/27[18:30] xi compile
/n/bleecker/u/bleecker/xi/Rivendell/Mtp/../shared/cproto -Dsun4 -Iincludes
/n/bleecker/u/bleecker/xi/Rivendell/Mtp/hfs/1024-1055/1052#1045#files#t_main.c/t
_main.c >
/n/bleecker/u/bleecker/xi/Rivendell/Mtp/hfs/1024-1055/1052#1045#files#t_main.c/t
_main.c.h.in
./envs/compile_local.env[159]:
/n/bleecker/u/bleecker/xi/Rivendell/Mtp/../shared/cproto:  not found
./envs/compile_local.env[169]: ../shared/config_files:  not found
./envs/compile_local.env[170]:
/n/bleecker/u/bleecker/xi/Rivendell/Mtp/../shared/gen_proto:  not found
gcc -Dsun4 -g -c -Wall -pedantic -ansi -Iincludes   -in: t_main.c
-out:t_main.c.o
compile successful, view results with viewErr
```

Figure 10. MTP activity completion.

extension language. *emacs*' extension language, called *E-Lisp*, allows users to define their own new functions and commands, and thus customize *emacs* to their applications.

Ad hoc *E-Lisp* functions were coupled with an augmented version of SEL, to effect a Grey Box integration, where the environment could perform loading of additional files into the same *emacs* instance at any time and discern which files had been updated. No special effort was required by the user, in contrast to the attention he/she must pay to MTP's pop-up window. This was achieved using one wrapper for the entire session, which dealt with addition of new buffers as new activities were submitted, rather than using a separate wrapper per activity. There was a major drawback to this approach, however: only one final status result could be returned to the environment, when *emacs* and its wrapper terminated, and all files were effectively recorded into the environment's repository at this same moment. In other words, it was not possible for the process to treat separately the different sets of data acquired throughout the work session—a central feature of MTP.

Later during the development of MTP, we looked at *E-Lisp* again to pursue Grey Box integration. Ad hoc *E-lisp* functions implemented a direct interface between *emacs* and the watcher utility, and also completely automated the initialization phase of the activities. The conclusion phase, particularly the choice of the `success` or `failure` return status for the separate activities run on the same instance of *emacs*, is still an explicit responsibility of the user even under this paradigm.

In general, `UNI_NO_QUEUE` appears appropriate for tools with some internal multi-tasking, multi-buffer or multi-context capability, but still not particularly useful or desirable for multi-user access. The main advantage of persistence for this class of tools is that the user can run partially ordered activities on the same instance of the program, without losing its intermediate state information, and possibly allowing for sharing or splicing (cut-and-paste) of intermediate results. Cut-and-paste can be intentionally directed *among* activities directed by the process, or even *within* a single activity that simultaneously presents multiple file arguments to the tool, in either case with the envelope's messages to the pop-up window instructing the user what to do. Note there is no means for preventing, from the environment, user-initiated cut-and-paste once the tool is designated as `UNI_NO_QUEUE`.

5.3. MULTI_QUEUE: FUF

FUF is a sophisticated unification-based tool running on top of Lisp and is used, among other things, in natural language processing research for the generation of sentences from corresponding syntactic data structures (Elhadad, 1993). It defines hierarchical procedures that apply in sequence one or more separate layers of unification rules to its input structures—as well as to the new structures produced by each step of the procedure—in order to obtain as output all the valid surface forms, under the constraints posed by the language rules. FUF is a typical Lisp-based interpreted application, in that it that supports various kinds of interactive tracing facilities and has the option to test and execute various data and program files, by loading and swapping them on the fly. As with most interpretive tools, it maintains sufficient information in memory to reflect the progress of its elaboration through the series of commands issued to it since start-up. Moreover, like many query systems constructed on top of Lisp, there is a long start-up time and it engages a considerable amount of system resources (notably main memory and swap space) and thus qualifies as a heavy-weight tool.

One of the main reasons for this choice as our exemplar `MULTI_QUEUE` tool is that it is easy to imagine a scenario in which, in order to process some data with FUF, multiple unification procedures are needed, each of which is the responsibility of a different member of a development group. Our paradigm could facilitate the testing and execution of the various phases of the project through a (modest) form of groupware: sequentially, each developer would load into FUF its own program, run it on the appropriate data and refine it as much as needed, and produce at the end an output that is also the input for the next step, also leaving the system in the correct state to begin the following activity. MTP moves the user interface among the users as they take their turns. The final outcome of the overall workflow would be produced by a single instance of the system and as the result of the collaboration of several users. Analogous collaborative work models could be applied to other programs, which outside the MTP framework could not be employed in this way. We have recently used the commercial *FrameMaker* word processing system in `MULTI_QUEUE` style; although it supports multiple buffers, it does not provide machinery for multiple users and thus GUI movement support is needed.

The envelopes we devised for this case study are devoted to loading within the memory of FUF a specific unification program, and to handle the correct system configuration for it, by asking the user to type the appropriate Lisp commands. The user might know little, if anything, about the configuration issues involved: he/she needs only to follow the instructions appearing in a pop-up window, since each envelope is specialized towards a separate portion of the group work. After this initial customization, the user is left completely free to query FUF and interact with it in the typical fashion of Lisp-based interpretive applications. Any files produced as result of these operations may be imported into the objectbase when the `success` choice ends the activity, as described above.

From a general point of view, the `MULTI_QUEUE` category allows the reuse of single instances of such computationally expensive programs throughout a series of activities. Another important point in favor of supporting this class is that the information retained in the tool's memory space (and not necessarily persistently on disk) represents both the current state of the system and the history of its past performance, and is generally necessary for generating the answer to new queries. This makes even more valuable the ability of the `MULTI_QUEUE` work model to support applications with long-duration work sessions that go beyond any individual process step, and to ensure common access to them to any set of users.

The most relevant consequence of the creation of this category is indeed that, by exploiting Activity Queues and the *xmove* facility that achieves passing of control over the user interface among users involved in a session, it allows us not only to conveniently integrate a vast and peculiar family of tools, but also to actually modify at the same time their intrinsic single-user nature and extend their use along the serial groupware lines described above. We consider this as one of the most interesting and meaningful results of this work.

5.4. MULTI_NO_QUEUE: Oz

We decided to use Oz itself as a testbench for the `MULTI_NO_QUEUE` category. The main reasons for this choice were the familiarity we have with Oz as a complete multi-user system and the in-house availability of the application in a ready-to-run state. Oz, as a typical client/server system (and unlike most applications based on peer-to-peer architectures), poses, in the most general case, the problem of treating differently the `OPEN-TOOL` command initiating a session, when it is necessary to start-up both the tool's server and a client, from those subsequently issued to join the session, which obtain further copies of only a client. Conversely, the last `CLOSE-TOOL` command in a session must deal with shutting down the tool's server. Moreover, since one can optionally employ a daemon that automatically starts up the Oz server with the first client and automatically shuts it down when the last client exits, Oz can also be used to simulate the behavior of non-hierarchical architectures, which do not need special treatment for the activation of its first and termination of its last components.

The intrinsic difficulties of dealing with these issues were solved in the context of the envelope indicated by the `path` field of the tool declaration and invoked by the `OPEN-TOOL` command. The designated envelope is invoked exactly once per session for all other categories of tools, but in the case of `MULTI_NO_QUEUE` is invoked separately for each user who joins the session—and thus must be able to, internally, distinguish its first from its subsequent invocations with respect to the same persistent tool. Oz's initialization envelope is shown in figure 6; this envelope handles the shut-down of Oz's server by invoking the auxiliary script given in figure 7. MTP, with its `MULTI_NO_QUEUE` class, is therefore able to support a generic multi-user tool, by forking and providing copies of the program to every participant in a session, as required by its structure.

MTP could easily be extended to allow for two distinct initialization envelopes in the `MULTI_NO_QUEUE` case, or in all cases—so that the first user to join a session and all subsequent users may be treated differently (of course the two scripts may be identical if no distinction is needed for the particular tool). Similarly, MTP could be extended to handle yet another separate envelope triggered by the `CLOSE-TOOL` command, or a pair of envelopes distinguishing between the last user to leave a session and all previously exiting users.

During our experiment with Oz, we devised MTP activities that perform operations within an in-progress workflow (the process state as well as the product data is persistent across sessions as well as tasks and activities within a session). Some wrappers instruct the user, with the usual pop-up messages, on how to use Oz's GUI to browse the objectbase, inspect the process definition task set, etc.; this could be useful for training new users. More significantly, it is also simple to ask users to initiate specific Oz tasks, or sequences of tasks. Alternatively, the MTP activity might simply instruct the user(s) as to *what* is to be accomplished, and leave it to the user(s) to determine *how* best to achieve that goal within the process supported by the MTP-invoked Oz instance (not to be confused with the MTP-invoking Oz instance).

This raises the possibility of an Oz meta-process that controls one (or more) Oz process(es), effecting a form of hierarchical workflow system. This could potentially address a certain limitation of Oz as a PCE, namely that relationships among tasks within a process are formed only with respect to satisfying local constraints, the task prerequisites and obligations, and there is no global topology or "grand view" (Kaiser et al., 1994). However, that grand view could feasibly be defined by the meta-process, by directing the workflow among abstract or at least aggregate tasks, while each MTP-invoked process itself directs only the workflow among concrete, perhaps primitive tasks, effectively filling in the details left out of the meta-process. The meta-process hierarchy could be elaborated to arbitrarily many levels, not just two. Further discussion of this idea is outside the scope of this paper.

There are some important differences between the integrations of collaborative tools, like Oz, and non-collaborative tools, which must be taken into account when considering the capabilities of the `MULTI_NO_QUEUE` work model. In the non-collaborative case, by definition each user is intended to be isolated from the rest and data access conflicts among overlapping argument sets are sporadic. In the case of data from the environment's repository, conflicts may be resolved before the arguments are passed to the tool by some concurrency control mechanism provided by the PCE; Oz, by default, implements conventional atomic and serializable transactions composed of individual or multi-step tasks (Heineman and Kaiser, 1995). When an external repository specific to the tool is employed (e.g., a database volume), the tool is assumed to have its own intrinsic concurrency control facilities.

In the collaborative case the issue of shared data becomes more problematic, even though most of the multi-user machinery is necessarily offered by the wrapped tool itself. A simple example is that of a multi-user editor (Dewan, 1993) invoked in the context of a groupware activity: the program itself permits and is able to deal with concurrent modification of its internal data, but from the viewpoint of environment's data repository it is necessary to support a concurrency control policy that allows multiple writers of the object containing the edited file attribute(s); this is achieved in Oz by defining and loading application-specific concurrency control policies, written in a notation (Heineman, 1996) that permits definition of extended transaction models including "cooperative transactions" (Kaiser, 1994). Concurrency control, per se, is not in the strictest sense part of the wrapping facility, but is nevertheless essential in order to fully integrate this class of tools. Further discussion of this topic is outside the scope of this paper.

6. Related work

As we pointed out in the previous sections, tool integration is of central importance to every effort to build efficient and practical software engineering support systems; therefore many studies have concentrated on defining and exploring the meaning and the dimensions of the term *integration* as applied to environments. Wasserman (1989), for example, identified five different kinds of integration:

- **Platform**. Concerned with interoperability among tools, achieved through the use of a common set of system services;
- **Presentation**. Stress on members of a toolkit giving the same "look and feel", via common GUI concepts and design;
- **Data**. Sharing data between different tools and handling the data relationships among objects produced by them;
- **Control**. Monitoring the tools' operation, and using such information to guide the development process; and
- **Process**. Realizing a well-defined software development process, by defining and tracking its steps.

According to this categorization, the work presented in this paper would be categorized mainly as control integration, even though guided by process.

In the attempt to fulfill the various requirements of control integration, and to overcome its inherent difficulties, the software engineering community has developed a wide spectrum of different approaches. Systems and methods are quite numerous, even when one decides—as we do in the rest of this section—to neglect what is probably the largest category: symbiotic *collections* of tools that (as, for instance, in the case of UNIX (Kernighan and Mashey, 1981)) are sometimes claimed as environments themselves, although they typically realize only platform integration.

Many methods embrace the White Box paradigm, with great variation among them with respect to the amount of tool code that must be generated or modified to achieve integration. An extreme approach in this sense is the realization of a set of custom tools, all managed by a common framework; typical and well-known examples are language-based environments generated by Gandalf (Habermann and Notkin, 1986) or the Synthesizer Generator (Reps and Teitelbaum, 1989), where usually tiny tool fragments are organized for execution in an *incremental* fashion as small portions of the program are edited, or interpretive systems such as Smalltalk (Goldberg and Robson, 1983), in which all the tools are combined together at run-time in the memory space of the language interpreter.

For many other environments, the common framework realizing a form of White Box integration of their toolset—focused on the data dimension—is represented by the database where the results of all the development activities, in their intermediate and final stages, are stored and shared. The tools are on the one hand forced to be closely related, since they must be able to use the same data formats, and on the other hand benefit in terms of performance, because they can reuse data produced by other utilities during previous operation. Some example databases intended for use by environments are GRAS (Kiesel et al., 1993), based on an extension of the classic Entity-Relationship data model, and Damokles (Dittrich et al., 1986), which employs schemas in the form of attributed graphs. Adele 2 (Belkhatir et al., 1991) enhances this methodology via a system of triggers connected to the state of the database, so that data modification by one tool is recognized and may cause the invocation of others.

The idea of assigning the role of main integration principle to a common object-oriented data repository has been employed quite widely, including by several of the projects aimed to define *standards* for building generic tools with a high degree of portability and interoperability, and therefore widely reusable—although only under the standard's specifications. PCTE (Gallo et al., 1989) is probably the best known of such standards. The goal of PCTE is to create a set of services and facilities, called a public tool interface, complete enough to support tool implementors in very different situations and domains; many environment prototypes and projects (Thomas, 1989; Bremeau, 1989; Georges and Keommer, 1989) already exploit this facility. Another proposed standard that exploits an object-oriented repository for its integration mechanism is the Ada-specific CAIS-A (Munck et al., 1989).

A different approach to the White Box paradigm, intended to be more cost-effective than building custom toolsets around a given framework, is represented by the class of systems based on *event notification*—whose stress is on control integration rather than data integration. Field (Reiss, 1990) is viewed by many as the archetype of this class of system: its basic principle is the addition of interface modules that send and receive specialized messages to the code of generic tools (in some cases this can be achieved by Grey Box extensions or Black Box wrappers). The messages produced by a tool are sent to a centralized component, known as the Broadcast Message Server (BMS), to inform it about the actions performed during the work session. The BMS elaborates them and produces further information that is sent on to other tools, who have registered for that pattern of message without necessarily any specific knowledge regarding the tool that produced it, in order to coordinate their operation.

YEAST (Rosenblum and Krishnamurthy, 1991) is another system using a form of event notification: it also has a client/server structure, in which the server accepts from the clients event pattern definitions associated with action specifications. It is also able to recognize the occurrences of events in the general computer system, such as time passage, timestamp modifications etc., or can be notified of such occurrences, either interactively by users or automatically by tools. In response to an event recognition, YEAST takes the actions that have been previously associated with that event.

Polylith (Purtilo, 1994) combines an event-driven approach with another technique in the spectrum of White Box integration: tool fragmentation. While entire external tools can be incorporated in Polylith, by relinking with the provided libraries that support the interface to the system's kernel, more often tools are identified with simpler *services*—or modules or subroutines—whose structure is declared in a service database, and whose free combination and communication is used to obtain the performance of various complex, full-fledged applications and to carry out all the tasks supported by the environment. Further, modules are configured in a distributed fashion, and may even be packaged up and moved among hosts during execution (Purtilo and Hofmeister, 1991). Many commercial *message bus* products, such as Sun Tooltalk, DEC FUSE and HP SoftBench, combine ideas introduced in Field and Polylith.

Tool fragmentation (usually in larger pieces than for the language-based editors above) is the basic integration principle of several systems, including RPDE (Harrison, 1987; Ossher and Harrison, 1990), Odin (Clemm and Osterweil, 1990) and IDL (Snodgrass and Shannon, 1986; Snodgrass and Shannon, 1990). RPDE maintains tables that represent its tool fragments as the cross-product of objects (i.e., structural components that can be manipulated by applications) and roles and methods (i.e., procedural components used to act upon objects). Odin has a very similar concept of objects and of the tool interactions that manipulate them; it also provides a language to specify tasks and composite tools, whose operators are represented by tool fragments and where objects play the role of their operands. Similarly, IDL proposes a notation to define the structural and functional features of its tools, each of which can be seen as a "building block" with a front-end for input, a composite structure defining its algorithm, and a back-end for its output. IDL declarative statements also describe how to connect several of these components into composite tools. The same kind of notation is now used as part of the CORBA distributed computing standard to describe data transmitted among clients and servers (Soley and Kent, 1994).

Since White Box, in all of its flavors, is the kind of integration most frequently implemented by environment builders, less work has been done on Grey Box methods. This paradigm does not require any code modification to the tools, which instead must provide an extension language or API, so that functions can be written to interact with the environment. Unfortunately, relatively few applications (aside from database management systems) are equipped with features that allow to build arbitrary functional interfaces to an environment framework. An attempt to address this limitation is presented by Notkin and Griswold (1988), who proposed a mechanism to dynamically and incrementally extend the functionality of generic software systems, without modifying the underlying source code.

Mediators have been proposed as a general architectural facility for integration of perhaps legacy applications whose interfaces do not nicely fit together and cannot readily be modified to match (Wiederhold, 1992). The mediators comprise special "glue" that make whatever transformations are necessary among relatively independent subsystems to make them work together, and often involve callbacks from the glue code to the application or vice versa—which assumes an API on the part of at least one of the several coupled components. This approach has been applied to large environment components such as object-oriented database management systems (Wells et al., 1992), transaction managers (Heineman and Kaiser, 1995), and process engines (Tong et al., 1994), as well as tools.

We maintain that Black Box integration, via tool wrapping/enveloping (a form of mediation without the explicit API and callbacks), is probably the most flexible and general methodology since its conceptual aim is the encapsulation in the environment of external tools with no changes to their code, nor need for other kinds of functional capabilities.

ISTAR (Dowson, 1987) appears to be the initiator of studies along these lines. While it provided its own development and integration toolkit to help construct new dedicated programs according to the needs of a particular environment, ISTAR also allowed use of third-party applications, simply by encapsulating their invocation into the code of ad hoc envelopes that provide the correct interaction with ISTAR's database and user interface.

As we already pointed out in Section 2, Oz employs shell-script envelopes to invoke the activities of process tasks and abstractly represents external application programs as object classes in a toolbase. Another example is offered by ProcessWEAVER (Fernström, 1993), a commercial system embracing Black Box integration and combining together a message bus and a process engine. ProcessWEAVER models tools as objects of class TOOL, and envelopes have the form of interpreted procedures with a syntax similar to UNIX shell scripts. Most process-centered environments, among those that do not rely on White Box methods, provide a system-specific enveloping language and /or exploit standard scripting languages such as Tcl (Ousterhout, 1990) or Python (Watters, 1995).

Many systems provide some means for off-loading the execution of tools away from where they would "normally" run. The simplest is remote job control, such as UNIX `rsh`, which invokes a program or script on a specified host. It can be used to take advantage of tools that do not operate on the user's machine. Some environments, such as Spice (Dannenberg, 1982) and DSEE (Leblang and Chase, 1987), automatically distribute tool executions to other hosts on a local area network. Their main goal is to achieve load balancing, e.g., for a large system build. These approaches seem limited to batch tools, such as compilers, with no user interaction. Batch tools inherently do not admit sharing of a single execution instance, except in the degenerate sense that multiple users may happen to want to compile the same version of a file and once is enough, but are easily amenable to Black Box integration methods.

WebMake (Baentsch et al., 1995) may be the ultimate combination of remote job control and load balancing, whereby tool invocations can be automatically sent over the Internet to other sites on the World Wide Web that participate in the WebMake protocol by installing a particular program (a "CGI-bin") in their website. The data might reside at a remote site, or the tool might need to execute on a particular machine architecture. Server load is considered, with the possibility of offloading to another host at the same site or back to the originating site, with all necessary data transfers handled transparently. Interactive tools can be invoked, but by delegating control to a resident user at the relevant Internet site rather than sending the GUI back to the originating user. We have recently constructed a Web-based Oz client (Dossick and Kaiser, 1996), which is intended to eventually support the same kind of facility.

Various systems support some form of tool instance sharing. XTV (Abdel-Wahab, 1994) is a utility related to *xmove*, but operating at a finer granularity and considerably more sophisticated. It displays the graphical user interface of an X Windows tool to multiple users *simultaneously*, as opposed to one at a time, but still only one user has control of the mouse and keyboard at any given moment. Tools may be integrated (with XTV, not a PCE) in Black Box fashion with no modification or extensions. If we had employed XTV instead of *xmove*[6], then most of our `MULTI_QUEUE` tools could nominally become `MULTI_NO_QUEUE` as far as MTP was concerned, but still lacking facilities for truly concurrent work. Suite (Dewan and Choudhary, 1992) is a toolkit for constructing shared GUIs for computer supported collaborative work tools, where generally the tools must be modified or written from scratch (i.e., White Box). It has been applied to a number of software engineering tools in Flecse (Dewan and Riedl, 1993). Suite also utilizes floor-passing, as in our `MULTI_QUEUE`, but with the advantage—like XTV—that all users can see the tool's GUI simultaneously.

7. Contributions and future work

We have fully implemented all the facilities discussed in this paper, except as noted in the text, and support the tools we chose as test cases for MTP's four work models. The completed experiments—all of which run quite satisfactorily—have demonstrated the feasibility of employing wrappers for persistent tools within a process-centered environment framework. We expect that an analogous approach would work for integrating legacy applications into a variety of software development environment frameworks and other kinds of integration architectures.

Further, we have introduced several useful concepts to the domain of Black Box tool integration, including a categorization of tools into families with diverse multi-user and multi-tasking capabilities, the notions of multiple complementary enveloping protocols and of loose wrapping, the idea of interfacing with already-executing persistent instances of programs external to the environment, and the ability to extend the functionality of intrinsically single-user tools to partial sharing of their data and computational resources. The support for directing tool execution to a proxy client, when the `host` or `architecture` field is non-empty, also extends to Oz's original SEL protocol, since the pragmatic problems of host licenses and platform dependencies apply even to the relatively mundane tools (compilers and the like) supported by previous approaches to Black Box enveloping.

The MULTI_NO_QUEUE model presented here is best suited to *asynchronous* groupware applications, where users enter and leave the tool as they please. There is as yet no facility in Oz to define, as part of the process, the circumstances under which tool sessions should be automatically opened /joined and exited /closed; adding such a feature would still allow for asynchronous groupware but more closely couple sessions with the workflow in a manner similar to how individual activities within those sessions are supported. We have already developed preliminary process support for *synchronous* groupware, in which multiple users perform an activity together at the same time (Ben-Shaul et al., 1994). For example, the `multi-flag` field, originally introduced for MTP, is now used within SEL to identify tools that support this kind of collaboration, so that the system can simultaneously submit the activity and its arguments to the clients corresponding to multiple designated users (Ben-Shaul and Kaiser, 1995). We have also recently added support for either a human user or the process to *delegate* control over pending tasks to alternative users (Tong et al., 1994), as opposed to machines, along with corresponding user interface support (agendas treated as menus to select which of the enabled tasks to do next).

One interesting future direction would be to split off all tool management (for both MTP and SEL) from the Oz server into a separate component, independent from the process engine, that would execute as another operating system process distinct from the server, user clients and proxy clients. This would lower the load on the server, simplify later replacement of the component within the Oz system (if desired), and ease the incorporation of both MTP and SEL facilities into other environment frameworks.

Acknowledgments

Prof. Kathy Mckeown provided the FUF application and served as the second reader for Mr. Valetto's Masters thesis (Valetto, 1994). Peter Skopp played a major part in designing and implementing the architectural changes needed to introduce proxy clients into Oz, a variant of which are used on a one-to-one basis to support low-bandwidth (modem) user clients (Skopp, 1995). George Heineman conducted the SEL Grey Box experiment involving emacs, and developed the watcher utility as part of that effort. Sonny Xi Ye is working with Prof. Kaiser on continued development of MTP, and generated the envelope scripts and screen dumps included in this paper. An extended abstract of this paper appears as Giuseppe Valetto and Gail E. Kaiser, Enveloping Sophisticated Tools into Computer-Aided Software Engineering Environments, *IEEE Seventh International Workshop on Computer-Aided Software Engineering*, July 1995, pp. 40–48.

Notes

1. The first use of the term "envelope" to refer to tool wrapping, that we know of, was with respect to the ISTAR system (Dowson, 1987).
2. SEL and many of the other Oz facilities mentioned in this paper were originally developed for our earlier system called MARVEL.
3. Proxy clients and user clients were initially referred to as Special Purpose Clients and General Purpose Clients, respectively (Valetto and Kaiser, 1995).
4. *idraw* takes about 15 elapsed seconds to start-up on a Sun SparcStation 10 workstation.

5. The second author has been known to keep the same *emacs* instance running for months, obviously persisting over numerous and often unrelated tasks.
6. We chose *xmove* over XTV primarily because the former was developed by another group at Columbia.

References

Abdel-Wahab, H.M. XTV. http:// www.cs.odu.edu/wash_cit/XTV/doc/xtv.html.

Baentsch, M., Molter, G., and Strum, P. 1995. WebMake: Integrating distributed software development in a structure-enhanced Web. In *3rd International World Wide Web Conference*, Darmstadt, Germany, Elsevier Science B.V. http.//www.igd.fhg.de/www/www95/proceedings/papers/51/WebMake/WebMake.html.

Barghouti, S.N. 1992. Supporting cooperation in the MARVEL process-centered SDE. In *5th ACM SIGSOFT Symposium on Software Development Environments*, H. Weber (Ed.), Tyson's Corner VA. Special issue of *Software Engineering Notes*, 17(5):21–31.

Belkhatir, N., Estublier, J., and Melo, W.L. 1991. Adele 2: A support to large software development process. In *1st International Conference on the Software Process: Manufacturing Compulese Systems*, M. Dowson, (Ed.), Redondo Beach CA, IEEE Computer Society Press, pp. 159–170.

Ben-Shaul, I.Z. 1991. An object management system for multi-user programming environments. Master's thesis, Columbia University, Department of Computer Science, CUCS-010-91.

Ben-Shaul, I.Z. Kaiser, G.E., and Heineman. G.T. 1993. An architecture for multi-user software development environments. *Computing Systems, The Journal of the USENIX Association*, 6(2):65–103, Spring.

Ben-Shaul, I.Z. and Kaiser, G.E. 1994. A paradigm for decentralized process modeling and its realization in the Oz environment. In *16th International Conference on Software Engineering*, Sorrento, Italy, IEEE Computer Society Press, pp. 179–188.

Ben-Shaul, I.Z., Heineman, G.T., Popovich, S.S., Skopp, P.D., Tong, A.Z., and Valetto, G. 1994. Integrating groupware and process technologies in the Oz environment. In *9th International Software Process Workshop: The Role of Humans in the Process*, C. Ghezzi, (Ed.), Airlie VA, IEEE Computer Society Press, pp. 114–116.

Ben-Shaul, I. and Kaiser, G.E. 1995. *A Paradigm for Decentralized Process Modeling*. Boston: Kluwer Academic Publishers.

Bremeau, C. 1989. The PCTE Contribution to Ada Programming Support Environments (APSE). In *Software Engineering Environments International Workshop on Environments*, F. Long (Ed.), of Lecture Notes in Computer Science, Chinon, France: Springer-Verlag, 467:151–166.

Chase, M. and Reubenstein, H. 1992. An assessment of KBSA and a look towards the future. Technical Report RL-TR-92-163, Rome Laboratory.

Clemm, G. and Osterweil, L. 1990. A mechanism for environment integration. *ACM Transactions on Programming Languages and Systems*, 12(1):1–25.

Dannenberg, R. 1982. *Resource Sharing In A Network Of Personal Computers*. PhD thesis, Carnegie Mellon University Department of Computer Science.

Dewan, P. (Ed.) 1993. *Special Issue on Collaborative Software, of Computing Systems, The Journal of the USENIX Association*. University of California Press, 6(2), Spring.

Dewan, P. and Choudhary, R. 1992. A high-level and flexible framework for implementing multiuser user interfaces. *ACM Transactions on Information Systems*, 10(4):345–380.

Dewan, P. and Riedl, J. 1993. Toward computer-supported concurrent software engineering. *Computer*, 26(1):17–27.

Dittrich, K.R., Gotthard, W., and Lockemann, P.C. 1986. Damokles—A database system for software engineering environments. In *Advanced Programming Environments*, of Lecture Notes in Computer Science, Springer-Verlag Berlin, 244:353–371.

Dossick, S.E. and Kaiser, G.E. 1996. WWW access to legacy client/server applications. In *5th International World Wide Web Conference*, Paris, France, pp. 931–940.

Dowson. M. 1987. ISTAR—An integrated project support environment. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, P. Henderson (Ed.), Palo Alto, CA, December 1986. Special issue of SIGPLAN Notices, 22(1):27–33.

Dowson, M. 1987. Integrated project support with ISTAR. *IEEE Software*, 4(6):6–15.

Elhadad, M. 1993. *Using Argumentation to Control Lexical Choice: A Unification-Based Implementation*, PhD thesis, Columbia University, Department of Computer Science.

Fernström, C. 1993. PROCESS WEAVER: Adding process support to UNIX. In *2nd International Conference on the Software Process: Continuous Software Process Improvement*, Berlin, Germany, IEEE Computer Society Press, pp. 12–26.

Gallo, F., Boudier, G.G., and Thomas, I. 1989. Overview of PCTE and PCTE+. *ACM SIGPLAN Notices*, 24(2).

Garlan, D. and Ilias, E. 1990. Low-cost, adaptable tool integration policies for integrated environments. In *4th ACM SIGSOFT Symposium on Software Development Environments*, R.N. Taylor (Ed.), Irvine CA, Special issue of Software Engineering Notes, 15(6):1–10.

Georges, M. and Koemmer, C. 1989. Use and extension of PCTE: The SPMMS information system. In *Software Engineering Environments International Workshop on Environments*, F. Long (Ed.), Lecture Notes in Computer Science, Chinon, France, Springer-Verlag, 467:271–282.

Gisi, M.A. and Kaiser, G.E. 1991. Extending a tool integration language. In *1st International Conference on the Software Process: Manufacturing Complex Systems*, M. Dowson (Ed.), Redondo Beach CA, IEEE Computer Society Press, pp. 218–227.

Goldberg, A. and Robson, D. 1983. *Smalltalk-80 The Language and its Implementation*. Reading, MA: Addison-Wesley.

Habermann, A.N. and Notkin, D. 1986. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, SE-12(12):1117–1127.

Harrison, W. 1987. RPDE^3: A framework for integrating tool fragments. *IEEE Software*, 4(6):46–56.

Heineman, G.T. 1996. *A Transaction Manager Component for Cooperative Transaction Models*, PhD thesis, Columbia University Department of Computer Science, CUCS-010-96. Forthcoming.

Heineman, G.T. and Kaiser, G.E. 1995. An architecture for integrating concurrency control into environment frameworks. In *17th International Conference on Software Engineering*, Seattle WA, ACM Press, pp. 305–313.

Heineman, G.T., Kaiser, G.E., Barghouti, N.S., and Ben-Shaul, I.Z. 1992. Rule chaining in MARVEL: Dynamic binding of parameters. *IEEE Expert*, 7(6):26–32.

Kaiser, G.E. 1994. Cooperative transactions for multi-user environments. In *Modern Database Systems: The Object Model, Interoperability, and Beyond*, W. Kim (Ed.), Chap. 20, ACM Press, New York, pp. 409–433.

Kaiser, G.E., Feiler, P.H., and Popovich, S.S. 1988. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40–49.

Kaiser, G.E., Popovich, S.S., and Ben-Shaul, I.Z. 1994. A bi-level language for software process modeling. In *Configuration Management*, W.F. Tichy (Ed.), Trends in Software, Chap. 2, John Wiley & Sons, (2):39–72.

Kaplan, S. (Ed.) 1993. *Conference on Organizational Computing Systems*, Milpitas CA: ACM Press.

Kaplan, S.M., Tolone, W.J., Carroll, A.M., Bogia, D.P., and Bignoli, C. 1992. Supporting collaborative software development with Conversation Builder. In *5th ACM SIGSOFT Symposium on Software Development Environments*, H. Weber (Ed.), Tyson's Corner VA. Special issue of Software Engineering Notes, 17(5):11–20.

Kernighan, B.W. and Mashey, J.R. 1981. The UNIX programming environment. *Computer*, 12(4):25–34.

Kiesel, N., Schurr, A., and Westfechtel, B. 1993. GRAS, a graph-oriented database system for software engineering applications. In *6th International Workshop on Computer-Aided Software Engineering*, H.-Y. Lee, T.F. Reid, and S. Jarzabek (Eds.), Singapore, pp. 272–286.

Krishnamurthy, B. and Barghouti, N.S. 1993. Provence: A process visualization and enactment environment. In *4th European Conference on Software Engineering*, Lecture Notes in Computer Science, Springer-Verlag, Garmisch-Partenkirchen, Germany, 717:151–160.

Leblang, D.B. and Chase, R.P. Jr. 1987. Parallel software configuration management in a network environment. *IEEE Software*, 4(6):28–35.

Munck, R., Oberndorf, P., Ploedereder, E., and Thall, R. 1988. An overview of the DOD-STD-1838A (proposed), the common APSE interface set, revision A. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, P. Henderson (Ed.), Boston MA, November 1988. ACM Press. Special issues of Software Engineering Notes, 13(5), November 1988 and SIGPLAN Notices, 24(2):235–247, February 1989.

Nicol, J.R., Wilkes, C.T., and Manola, F.A. 1993. Object orientation in heterogeneous distributed computing systems. *Computer*, 26(6):57–67.

Notkin, D. and Griswold, W.G. 1988. Extension and software development. In *10th International Conference on Software Engineering*, Raffles City, Singapore, IEEE, pp. 274–283.

Ossher, H. and Harrison, W. 1990. Support for change in RPDE^3. In *4th ACM SIGSOFT Symposium on Software Development Environments*, R.N. Taylor (Ed.), Irvine CA, ACM Press. Special issue of SIGSOFT Software Engineering Notes, 15(6):218–228.

Ousterhout, J.K. 1990. Tcl: An embeddable command language. In *Winter 1990 USENIX Conference*, Washington DC, USENIX Association, pp. 133–146.

Popovich, S.S. 1992. Rule-based process servers for software development environments. In *1992 Centre for Advanced Studies Conference (CASCON)*, J. Botsford, A. Ryman, J. Slonim, and D. Taylor (Eds.), Toronto ON, Canada, IBM Canada Ltd. Laboratory, I:477–497.

Purtilo, J.M. 1994. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174.

Purtilo, J.M. and Hofmeister, C.R. 1991. Dynamic reconfiguration of distributed programs. In *11th International Conference on Distributed Computing Systems*, Arlington TX, IEEE Computer Society Press, pp. 560–571.

Reference Model for Framework of Software Engineering Environments: Edition 3 of Technical Report ECMA TR/55. 1993. NIST Special Publication 500-211. Available as /pub/isee/sp.500-211.ps via anonymous ftp from nemo.ncsl.nist.gov.

Reiss, S.P. 1990. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66.

Reps, T.W. and Teitelbaum, T. 1989. *The Synthesizer Generator A System for Constructing Language-Based Editors*. Texts and Monographs in Computer Science. New York: Springer-Verlag.

Rosenblum, D.S. and Krishnamurthy, B. 1991. An event-based model of software configuration management. In *3rd International Workshop on Software Configuration Management*, P.H. Feiler (Ed.), ACM Press, pp. 94–97.

Skopp, P.D. 1995. Low bandwidth operation in a multi-user software development environment. Master's thesis, Columbia University Department of Computer Science, CUCS-035-95.

Snodgrass, R. and Shannon, K. 1986. Supporting flexible and efficient tool integration. In *Advanced Programming Environments*, T.M. Didriksen, R. Conradi, and D.H. Wanvik (Eds.), Lecture Notes in Computer Science, Springer-Verlag, Trondheim, Norway, 244:290–313.

Snodgrass, R. and Shannon, K. 1990. Fine grained data management to achieve evolution resilience in a software development environment. In *SIGPLAN '90 4th ACM SIGSOFT Symposium on Software Development Environments*, R.N. Taylor (Ed.), Irvine CA. Special issue of SIGSOFT Software Engineering Notes, 15(6):144–156.

Soley, R.M. and Kent, W. 1994. The OMG object model. In *Modern Database Systems: The Object Model, Interoperability, and Beyond*, W. Kim (Ed.), Chap. 2, ACM Press, New York NY, pp. 18–41.

Solomita, E., Kempf, J., and Duchamp, D. 1994. Xmove: A pseudoserver for X window movement. *The X Resource*, 1(11):143–170.

Stallman, R.M. 1981. Emacs the extensible, customizable, self-documenting display editor. In *SIGPLAN SIGOA Symposium on Text Manipulation*, ACM, Special issue of SIGPLAN Notices, 16(6):147–156.

Thomas, I. 1989. PCTE interfaces: Supporting tools in software-engineering environments. *IEEE Software*, 6(6):15–23.

Thomas, I. 1989. Tool integration in the PACT environment. In *11th International Conference on Software Engineering*, Pittsburgh PA, IEEE Computer Society Press, pp. 13–22.

Tong, A.Z., Kaiser, G.E., and Popovich, S.S. 1994. A flexible rule-chaining engine for process-based software engineering. In *9th Knowledge-Based Software Engineering Conference*, Monterey CA, IEEE Computer Society Press, pp. 79–88.

Transcending Boundaries: ACM 1994 Conference on Computer Supported Cooperative Work. Chapel Hill, NC: ACM Press.

Valetto, G. 1994. Expanding the repertoire of process-based tool integration. Master's thesis, Columbia University, Department of Computer Science, CUCS-027-94.

Valetto, G. and Kaiser, G.E. 1995. Enveloping sophisticated tools into computer-aided software engineering environments. In *IEEE 7th International Workshop on Computer-Aided Software Engineering*, Toronto Ontario, Canada, pp. 40–48.

Vlissides, J.M. and Linton, M.A. 1990. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268.

Wasserman, A.I. 1989. Tool integration in software engineering environments. In *Software Engineering Environments: International Workshop on Environments*, F. Long (Ed.), Lecture Notes in Computer Science, Chinon, France, Springer-Verlag, 467:137–149.

Watters, A.R. 1995. The what, why, who, and where of Python. *UnixWorld Online*, Tutorial Article No. 005.

Wells, D.L., Blakeley, J.A., and Thompson, C.W. 1992. Architecture of an open object-oriented database management system. *Computer*, 25(10):74–82.

Wiederhold, G. 1992. Mediators in the architecture of future information systems. *Computer*, 25(3):38–49.

Automated Software Engineering 3, 347–367 (1996)
© 1996 Kluwer Academic Publishers. Manufactured in The Netherlands.

Use of Methods and CASE-Tools in Norway: Results from a Survey

JOHN KROGSTIE johnkrog@idt.unit.no
Norwegian Institute of Technology (NTH), Faculty of Electrical Engineering and Computer Science, UNIT/NTH, IDT, N-7034 Trondheim, Norway

Abstract. Results on use of methodology and CASE-tools from a survey investigation performed in Norwegian organizations are presented. The results are based on responses from 52 Norwegian organizations on a survey investigation on development and maintenance.

Although there appears to be a trend towards the use of more packaged solutions, the investigation indicates a larger proportions of application systems being developed as customized systems in larger organizations. The presence of a comprehensive development and maintenance methodology and the use of CASE-tools are also more prominent in larger organizations. Larger organizations also use statistically significant less of their effort on functional maintenance. Even though, the impact of CASE-tools on the information systems portfolios of Norwegian organizations are not yet large, and improvements in functional maintenance can not be attributed to the use of CASE. A notably different perception on the benefit of CASE-technology for productivity was observed between users and non-users of CASE, but the difference was not found to be statistically significant.

Keywords: CASE, methodology, survey investigation

1. Introduction

In this article, we present some of the results from a survey-investigation being performed in Norwegian organizations during the summer of 1993 on development and maintenance of computerized information systems. The results being presented in the article deal mostly with development manner, development methodology, use of organizational controls, and the use of CASE-tools. Earlier papers have presented result from the survey on more maintenance related topics (Krogstie, 1994a; Krogstie and Sølvberg, 1994). A comprehensive report from the investigation is also available (Krogstie, 1994b; Krogstie, 1995a). Throughout the paper, comparisons are made with the results from other investigations. Whereas these investigations primarily present descriptive results, we have included further statistical analysis. None of the earlier investigations present an integrated investigation of the use of CASE-tools and the use of a comprehensive development and maintenance methodology.

Section 2 describes our research method. Section 3 describes the dependant variables we use in the analysis, whereas the main results are presented in Section 4. Section 5 contains a summary of the results.

2. Research method

Two forms, one containing questions regarding the development and maintenance practice followed in the organization as a whole, and one with questions regarding the maintenance of one application system, were distributed by mail to 350 Norwegian organizations. Most of the results being reported in this article are based on the responses to the first form. The organizations were taken from the list of member organizations of DND (Den Norske dataforening), the Norwegian Computer Society. This population implicitly assured that the forms were answered by organizations of some size which look upon computerized information system support as important for their business. Whereas the average number of employees of Norwegian organizations within the areas of manufacturing, industry, trade and services is 8 persons (Central Bureau of Statistics of Norway, 1992), the similar average among the respondents of our investigation was 2347, thus our respondents are above average when it comes to size according to Norwegian standards.

The design of the forms (Krogstie, 1994b) was based on forms used in previous investigations within the area of development and maintenance of computerized information systems, in particular (Lienz and Swanson, 1980; Swanson and Beath, 1989; Henne, 1992). In addition, a set of questions regarding the use of CASE-tools were included. The forms were refined through several pilot fill outs before being distributed on a large scale.

On some of the questions, it was possible for the respondents to assess the quality of the answers. There was also room for issuing additional remarks on many of the questions. This, in addition to the general design of the forms enhanced the cross-checking of the responses. Where discrepancies were discovered, these were further investigated by phone. No rigorous post-investigation validation similar to what was done in Arnold and Parker (1982) was performed.

A total of 78 answers were returned, giving a response rate of 22%. Some of the answers were negative, replying that the organization was not doing work of the sort which was queried about. Other answers also had to be dismissed, giving us a total of 52 valid answers as a basis for analysis. SPSS (Norusis, 1992) has been used for all statistical analysis of the data. According to the distribution of the variables, statistical significance has for hypothesis testing been determined either by using the twin-tailed Student t-test, or the twin-tailed Mann-Whitney test. In correlation analysis, Pearson's correlation coefficient or Spearman's rank has been used as appropriate. Similar to (van Swede and van Vliet, 1994), we apply a r-level of 0.25 with a significance level of 0.05. Traditional tests for normality of distributions to decide on the use of further analysis-methods were also applied (Norusis, 1992).

2.1. *Hypothesis*

The following null-hypotheses were formulated based on the literature and experience:

- H1: There is no statistically significant difference between the maintenance effort in organizations that have a comprehensive development methodology and those that don't.
- H2: There is no statistically significant difference between the functional maintenance effort in organizations that have a comprehensive development methodology and those that don't.
- H3: There is no statistically significant difference between the size of the organizations that have a comprehensive development methodology and those that don't.

- H4: There is no statistically significant difference between the maintenance effort in organizations that use CASE-tools in development and maintenance and those that don't.
- H5: There is no statistically significant difference between the functional maintenance effort in organizations that use CASE-tools in development and maintenance and those that don't.
- H6: There is no statistically significant difference between the size of the organizations that use CASE-tools in development and maintenance and those that don't.
- H7: There is no statistically significant difference between the size of organizations when it comes to how they develop their major systems (customized or packaged development).
- H8: There is no statistically significant difference in the perception of the benefits of CASE between users and non-users of CASE-tools.
- H9: The institutionalization of organizational controls do not influence the behavior during maintenance significantly.

3. Background information

The respondents to the survey were in: Manufacturing and industry (20), public services (7), insurance and banking (8), trade (6), other areas (11). 80% of the organizations had a yearly data processing budget above 5 mill. Nkr. The forms were filled out by persons with long experience with information systems related work (median 18 years).

Work on computerized information system was divided into six categories in the survey:

1. Correcting errors in systems in production.
2. Adaption to a changed technical architecture.
3. Developing new functionality in existing systems.
4. Improving non-functional properties, e.g., performance of existing systems.
5. Developing new systems with functionality similar to the one found in old systems.
6. Developing new systems in new functional areas.

The four first categories are traditionally classified as maintenance activities, whereas the last two are development activities. Functional development consists of work in category 3 and 6, whereas functional maintenance consists of work in category 1, 2, 4, and 5. A deeper discussion on the usefulness of the distinction of functional maintenance and development which are one of the areas which differentiate this investigation from previous investigations of this sort is given in Krogstie (1995b).

3.1. Dependant variables

When investigating our hypothesis, we used the following dependant variables:

1. The percentage of the complete effort on development and maintenance of the application systems portfolio that is used on maintenance.
2. The percentage of the complete effort on development and maintenance of the application systems portfolio that is used on functional maintenance.

3. The logarithm of the number of employees in the organization.
4. The logarithm of the number of employees in the data department.
5. The logarithm of the number of systems developers in the data department.
6. The logarithm of the number of major application systems in the organization.
7. The logarithm of the number of end-users of the application systems.

Whereas the two first indicate the distribution of effort, the five last are different indicators of size of the organizations and their computerized information systems support. The distributions of these variables as reported were skewed to the left. Skewed distributions are quite regular in software-related research and call for very careful interpretations of findings (Dekleva, 1992c). A conversion to a logarithmic form is often used, as also applied by us. Such conversions normalizes the distribution, which is required for parametric statistical testing.

Descriptive statistics of the dependant variables are given in Table 1, whereas tests for normality for the normalized figures are given in Table 2.

The results presented in Table 2 do not give us any reason to reject the null-hypothesis that the numbers for both traditional maintenance and functional maintenance are normally distributed. This is also the case for the size measures, except for on the number

Table 1. Descriptive data for dependant variables.

Figure	Responses	Range	Mean	Median	SD
Maintenance	48	[10–100]	58.6	63.3	24.17
Functional maintenance	46	[10–90]	43.9	46.7	17.89
Number of employees	52	[20–35000]	2347	555	6499.54
Employees in data department	52	[1–250]	24.3	10	40.64
Number of systems developers	52	[0–87]	9.5	5	16.33
Number of major systems	51	[2–100]	10.3	5	18.86
User population	50	[20–5000]	541	250	883.80

Table 2. Tests for normality of dependant variables.

Figure	Skewness	Kurtosis	Shapiro wilks	Sign.	Lilliefors (K-S)	Sign.
Maintenance	−.1111	−.8133	.9559	.1386	.0706	>.200
Functional maintenance	.1513	−.0974	.9728	.4770	.0701	>.200
Log(employees)	.5250	.3469			.0897	>.200
Log(data department)	.1492	.1174			.0763	>.200
Log(systems developers)	.5724	.5284	.9507	.0796	.0954	>.200
Log(major systems)	.3896	.5085	.9489	.0618	.1332	.0296
Log(user population)	.1979	−.1786	.9774	.6236	.0624	>.200

Table 3. Correlating size and effort measures.

Figure	*r*	*N*	*p*
Log(number of employees)			
Log(Data department)	**.6705**	52	**.000**
Log(systems developers)	**.6137**	47	**.000**
Log(major systems)	**.5034**	49	**.000**
Log(user population)	**.5623**	50	**.000**
Maintenance	−.1616	48	.272
Functional maintenance	−.2685	46	.071
Log(data department)			
Log(systems developers)	**.8854**	47	**.000**
Log(major systems)	**.3841**	49	**.006**
Log(user population)	**.6974**	50	**.000**
Maintenance	**−.3496**	48	**.015**
Functional maintenance	**−.4487**	46	**.002**
Log(system developers)			
Log(major systems)	**.4407**	44	**.003**
Log(user population)	**.7003**	45	**.000**
Maintenance	**−.2949**	45	**.049**
Functional maintenance	**−.3115**	43	**.042**
Log(major systems)			
Log(user population)	**.3306**	47	**.023**
Maintenance	−.1874	46	.212
Functional maintenance	−.2619	45	.082
Log(user population)			
Maintenance	**−.2511**	46	.092
Functional maintenance	**−.2509**	44	.100
Maintenance			
Functional maintenance	**.4000**	46	**.006**

of major application systems. For this, only non-parametric tests are used. None of the distributions are perfectly normal, since the kurtosis and skewness are different from zero, but this would be expected even for a sample from a normal distribution (Norusis, 1992).

Inter-correlations of the variables are given in Table 3.

Significant figures are shown in **boldface**. We will not discuss the connection between the size and effort measures in detail in this article. This is done in (Krogstie, 1995b).

4. Results and discussion

The results are divided into three section:

- System development and maintenance methodology.
- Application of CASE-tools.
- Perceptions on the benefits of CASE-tools.

4.1. System development methodology

The development background for the major application systems in the responding organizations is illustrated in figure 1.

In another Norwegian investigation (Bergersen, 1990) 58% of the projects reported upon were own development, 27% were packages with large adjustments and 8% were packages with small adjustments (7% were a combination of two of these three).

The distribution of development background from the Swanson/Beath investigation (Swanson and Beath, 1989) is for comparison illustrated in figure 2. The most notable difference between the two Norwegian investigations and Swanson/Beath is in the number of packaged systems being used. The greater percentage of packages is probably due to that our organizations are on the average smaller, and as such do not find it cost-efficient in all cases to develop customized solutions. (Swanson/Beath reported in their case-study a median of 102 persons working in the data department, range [7–266], mean 95). Another possible explanation is that it now exist better customizable packages than it did in the late eighties. On the other hand did we not find any significant correlations between the dominant development manner and the age distribution of the portfolio to support this. The correlations between the percentage of the portfolio that is developed using the different development methods and the size-measures are given in Table 4. The Spearman rank is used. Since the number of systems being developed by the user organization was so small, we do not show these figures (none of the correlations were significant).

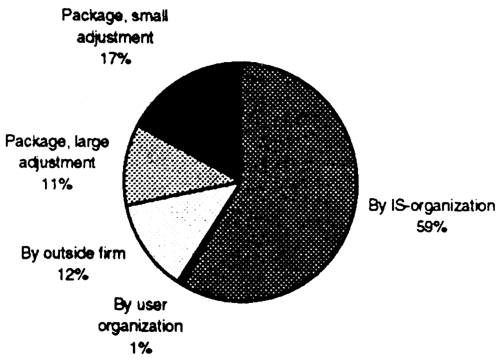


Figure 1. Development background of portfolio.

Table 4. Development method vs. size.

Figure	*r*	*N*	*p*
	Percentage	developed	by IS-organization
Number of employees	.2265	50	.114
Size of data department	**.6016**	50	**.000**
Number of system developers	**.6968**	50	**.000**
Number of major system	.2498	50	.080
Number of end-users	**.4588**	48	**.001**
	Percentage	developed	by outside firm
Number of employees	**−.2505**	50	.079
Size of data department	**−.3222**	50	**.023**
Number of system developers	**−.3165**	50	**.025**
Number of major system	.1381	50	.339
Number of end-users	−.1908	48	.194
	Percentage	package	large internal adjustments
Number of employees	.2398	50	.093
Size of data department	.2237	50	.118
Number of system developers	.1577	50	.274
Number of major system	.1619	50	.261
Number of end-users	.0251	48	.865
	Percentage	package	small internal adjustments
Number of employees	−.1783	50	.215
Size of data department	**−.3250**	50	**.021**
Number of system developers	**−.3313**	50	**.019**
Number of major system	**−.3935**	50	**.005**
Number of end-users	**−.2661**	48	.067

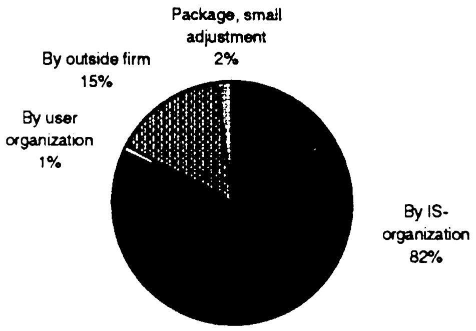


Figure 2. Development background of portfolio in (Swanson and Beath, 1989).

Inter-correlating the development manner percentages gave that the percentage of systems being developed by the IS-organization was negatively correlated with the percentage of packages with small internal adjustments ($r = -\mathbf{.5481}$, $N = 50$, $p = \mathbf{.000}$), and so was the percentage of systems developed by an outside firm ($r = -\mathbf{.2901}$, $N = 50$, $p = \mathbf{.041}$).

From the results in Table 4 it appears that organizations with larger data departments, number of systems developers, and number of end-users have a larger percentage of customized application systems, whereas larger organizations with large data departments have a smaller proportion of application systems made by outside firms. A large percentage of packages with small adjustments is more often found in organizations with small IS-departments and few end-users. Based on this, we reject hypothesis H7.

Before investigating on the use of an overall methodology we give an overview of the use of organizational controls for handling of change requests and maintenance of the portfolio. An overview over the use pattern is given in Table 5.

A comparison with earlier investigations (Lientz and Swanson, 1980; Nosek and Palvia, 1990) was given in Krogstie and Sølvberg (1994) and showed that the pattern of organizational controls was somewhat different. Some areas, like cost justification, retesting of changes and batching of changes appears to be better taken care of in Norway than in America. On the other hand, such controls as logging of user request, logging of changes and performing periodic formal audits seem to be better taken care of in the American organizations. The use of charge-backs also seems to be smaller. On the other hand, if we look at how the organizational controls are used in the maintenance of individual systems, we find that for instance only 32% answered 5 or 4 on a scale from 5 to 1 where 5 indicate *always* and 1 *never* on the question **Is the consequences of changes properly assessed?** Comparing this with the answer of 54% of organizations saying that all user request for changes to the application system must be cost justified shows a rather high discrepancy. This pattern is general, the application of many organizational controls are assessed to be better than how they are actually used.

Table 5. Application of organizational controls.

	Control	Use
1.	All user requests logged	77%
2.	All user requests cost justified	54%
3.	All changes logged and documented	67%
4.	All changes are formally re-tested	79%
5.	Changes are batched for periodic implementation	40%
6.	A formal audit is made periodically	8%
7.	Equipment costs are charged back to the user	40%
8.	Personnel costs are charged back to the user	31%
9.	Change requests are classified	60%
10.	Update of documentation is ensured	25%
11.	Users are informed of the status of their CRs	79%
12.	The same routines are used for all CRs	58%

To check if the organizational controls actually do have an impact on behavior during maintenance, we have split up the answers to certain questions in Part II of the investigation according to if the organizational control is supported or not. The results are given in Table 6. The first number refers to the organizational control, the second is the question used in Part II, the third shows the mean value for those not using the control, whereas the fourth indicates the mean value for those having the control institutionalized. The last column indicates the statistical significance for accepting the hypothesis that the figures from those having and those not having the controls institutionalized are equal.

The areas where the controls do not seem to have a significant impact on the behavior are on keeping the users informed about change requests and on updating the documentation of the programs. Apart from this we see that when comparing the work on separate systems compared to the institutionalization of organizational controls, that these seem to influence this work in most cases, rejecting H9.

When comparing the numbers for organizational controls further, we have divided them up in three areas:

1. Handling change requests: This included controls 1, 2, 9, 11, and 12.
2. Updating the existing application system: This includes controls 3, 4, 5, 6, and 10.
3. Charge-back of maintenance-costs: This includes controls 7 and 8.

Descriptive statistics for these areas are given in Table 7.

Table 6. Connection between use and institutionalization of organizational control.

Control	Question from part II	No control	Control	*p*
1	Is an overview of who is asking for changes kept?	3.3	4.2	.065
	Is statistics over this made?	1.8	2.0	.712
2	Is the consequences properly assessed?	2.6	3.6	**.001**
	Is time and cost assessed for an individual change?	2.6	3.3	**.038**
9	Is maintenance tasks categorized by type?	2.1	2.9	.067
	Is maintenance tasks categorized by importance?	3.7	4.3	.091
10	When changing a program, is documentation updated?	3.2	3.6	.243
11	Is the users informed on the status of CRs	3.7	3.9	.583
12	Do all changes undergo the same kind of control	3.5	4.2	**.049**

Table 7. Descriptive data for organizational controls.

Figure	Responses	Range	Mean	Median	SD
Organizational controls	48	[1–10]	6.188	7	2.647
Change requests controls	48	[0–5]	3.292	4	1.473
Maintenance controls	48	[0–4]	2.187	2	1.232
Charge-backs	48	[0–2]	.708	0	.898

The distributions of these variables were all found to be non-normal thus Spearman's rank is used in the further statistical analysis.

A comparison of the use of organizational controls and the dependant variables is given in Table 8.

Overall, the use of organizational controls is more usual in organizations with large data departments and many developers, and where there are many end-users. Charge-backs follow the same pattern, whereas only the size of the data department is significantly correlated with the maintenance controls, and none with change-request related controls.

31% of the respondents reported that they used a comprehensive development methodology covering all tasks of development and maintenance. Due to the small number of users of individual methodologies we do not differentiate between different methodologies below. All organizations having a complete development methodology had a yearly IS-budget of more than 5 million Nkr. The descriptive statistics for the number of employees in these organizations were:

Range [70–35000], Mean 5858, Median 1150.

Compared to the average, we see that overall it is organizations of some size that have a complete development methodology installed.

Results from dividing the sample according to if the organization use a complete development method or not and investigating on the connection with our dependant variables are given in Table 9.

Whereas all the size measures except the portfolio-size were significant, only functional maintenance of the maintenance measures were significantly smaller in the organizations having a complete development methodology. Based on this, we reject hypothesis H2, but do not have grounds for rejecting hypothesis H1. We also reject hypothesis H3.

In Table 10 the results of a similar non-parametric test are given investigating the connection between having a complete development methodology and the development manner and the use of organizational controls.

As expected, we see that the use of organizational controls including the use of maintenance controls and charge-backs are significantly higher in organizations claiming to have a comprehensive development and maintenance methodology. The same is not the case for the use of change request-controls. Those having a comprehensive development methodology have a significantly larger proportion of their application systems being customized, and a significantly smaller proportion of packages with small adjustments, which is neither unexpected comparing to Table 4.

4.2. Application of CASE-tools

CASE-technology was used by 27% of the organizations for development and 10% of the organizations for maintenance. All organizations using CASE had a yearly IS-budget of more than 5 million Nkr. The descriptive statistics for the number of employees of the CASE-users were:

Range [150–35000], Mean 6645, Median 1000.

Table 8. Use of organizational controls vs. size and effort measures.

Figure	*r*	*N*	*p*
Organizational controls			
Number of employees	.2059	48	.160
Size of data department	**.3812**	48	**.008**
Number of system developers	**.3232**	44	**.032**
Number of major system	.1879	46	.211
Number of end-users	**.3316**	47	**.023**
Maintenance	−.0634	46	.676
Functional maintenance	−.1899	44	.217
Change requests			
Number of employees	−.0975	48	.510
Size of data department	.2312	48	.114
Number of system developers	.1636	44	.289
Number of major system	−.0604	46	.690
Number of end-users	.2393	47	.105
Maintenance	−.0774	46	.609
Functional maintenance	−.2018	44	.189
Maintenance controls			
Number of employees	−.0327	48	.826
Size of data department	**.3016**	48	**.037**
Number of system developers	.2078	44	.176
Number of major system	.1417	46	.347
Number of end-users	.1319	47	.377
Maintenance	−.0157	46	.918
Functional maintenance	−.1349	44	.383
Charge-backs			
Number of employees	**.4587**	48	**.001**
Size of data department	.2450	48	.093
Number of system developers	.2088	44	.174
Number of major system	**.4018**	46	**.006**
Number of end-users	**.3369**	47	**.021**
Maintenance	−.0185	46	.903
Functional maintenance	−.0340	44	.827

Table 9. Maintenance effort and size vs. comprehensive development method.

	Yes			No				
Figure	*N*	Mean	SD	*N*	Mean	SD	Δ	*p*
Maintenance	14	50	25.929	33	60.9	22.215	−10.9	.150
Functional maintenance	14	34.2	13.947	31	47.1	17.253	−12.9	**.018**
Log(employees)	15	3.0834	.792	34	2.4813	.556		**.004**
Log(data department)	15	1.4526	.464	34	.8697	.451		**.000**
Log(system developers)	15	.9901	.471	30	.6204	.357		**.005**
Log(major systems)	14	.8266	.228	33	.7118	.254		.151
Log(user population)	15	2.6499	.548	32	2.2651	.463		**.016**

Table 10. Development manner and use of organizational controls vs. comprehensive development method.

Figure	Δ	*p*
Percentage developed by IS-organization	+	**.011**
Percentage developed by outside firm	−	.208
Percentage package, large internal adjustment	+	.229
Percentage package, large internal adjustment	−	**.006**
Organizational controls	+	**.007**
Change request controls	+	.263
Maintenance controls	+	**.026**
Charge-backs	+	**.004**

Compared to the average, we see that overall it is organizations of some size that uses CASE-technology. The CASE-users were from different areas, such as banking, construction, packing, trade, interest organization, government, transportation, and media. Most notably, all the respondents within the oil-industry (5) were applying CASE-tools supporting a comprehensive development methodology.

10 out of the 13 organization using CASE-technology (77%) applied a comprehensive development methodology. The CASE-users not having a comprehensive development methodology were the smallest organizations in terms of employees that used CASE. We would thus expect somewhat the same pattern as on development methods above. Table 11 gives an overview.

We see that it was basically large organizations who had adopted CASE-tools thus rejecting hypothesis H6. Even if the amount on maintenance and functional maintenance was less when using CASE-tools, (although not significant), we do not believe that it is the use of CASE that is the main reason for this, but rather that CASE is being taken into use in large organizations with an already developed development and maintenance methodology. For the long-term success of the application of CASE-tools this seems promising, since it is generally regarded as necessary to have a comprehensive development methodology

Table 11. Maintenance effort and size vs. use of CASE.

	Yes			No				
Figure	*N*	Mean	SD	*N*	Mean	SD	Δ	*p*
Maintenance total	12	49.4	25.416	34	61.5	23.970	−12.1	.143
Functional maintenance	11	35.4	17.036	33	46.8	17.788	−11.4	.069
Log(employees)	13	3.1626	.786	35	2.4468	.545		**.001**
Log(data department)	13	1.4351	.511	35	.9030	.418		**.002**
Log(system developers)	13	1.0088	.560	32	.6209	.336		**.006**
Log(major systems)	11	.9488	.240	35	.6805	.220		**.001**
Log(user population)	13	2.6572	.540	33	2.2906	.476		**.029**

Table 12. Development manner and use of organizational controls vs. use of CASE.

Figure	Δ	*p*
Percentage developed by IS-organization	+	**.014**
Percentage developed by outside firm	−	.989
Percentage package, large internal adjustment	+	.428
Percentage package, large internal adjustment	−	**.000**
Organizational controls	+	**.041**
Change request controls	+	.443
Maintenance controls	+	.085
Charge-backs	+	**.016**

in widespread use before applying CASE-tools supporting this methodology (Parkinson, 1990; Stobart et al., 1993). We will discuss this further below, but have no support for rejecting hypothesis H4 and H5.

In Table 12 a similar comparison is given with development manner and use of organizational controls.

Also here we find a similar pattern as for the use of a comprehensive methodology, with the exception of the use of maintenance control which are not significantly higher among the CASE-users.

A similar investigation in England reported that 18% of the respondents were using CASE-tools in 1990 (Stobart et al., 1991) whereas 26% were evaluating the introduction of CASE at that time. In a follow-up study in 1994 (Hardy et al., 1995), 43% of the organizations were currently using CASE-tools. Size-measures for these organizations were not given in Hardy et al. (1995), Stobard et al. (1991), thus it is difficult to perform a more detailed comparison.

It does not seem that any single CASE-tool has a dominant position in Norway, like for instance SDW has had in Holland for several years (Kusters and Wijers, 1993; Wijers and van Dort, 1990). No tool was reported to be used by more than two organizations, and a total of 11 different CASE-tools were mentioned. Neither in the UK is there any CASE-tool that is predominant (Hardy et al., 1995).

The average experience with the CASE-tools that was currently applied was 2.8 years, ranging between a half and 8 years. 5 of 12 (42%) had used the CASE-tool for more than 2 year. In an investigation from Holland from 1989 (Wijers and van Dort, 1990) the similar number was 11% , whereas in the investigation from 1992 (Kusters and Wijers, 1993), 53% of the users had more than two years experience. From this it seems that application of CASE may have come further in other countries, even if it is not possible to be certain about this from the data we have available.

The CASE-tools supported between 1 and 4 of the major application systems in the organizations (mean 1.7). Compared to the number of major systems in these organization, we had that 21 out of 214 application systems were supported by CASE (10%). Looking on the individual organization, the number of application systems supported by CASE varied between 0 and 50% of the major systems, with a mean of 19%. Some of the comments given by the respondents indicated that CASE has just recently been put to practical use, and that CASE in many places is used to support only parts of application development and maintenance.

Based on this, we feel that it is too early to assess the overall importance of CASE on a portfolio level, since it seems to have had little influence on overall systems development and maintenance this far in most organizations. The results above on smaller amount of effort on maintenance and functional maintenance among CASE-users are probably linked to the size of the organizations and the presence of a development methodology.

Table 13 shows the usage-areas of the CASE-tools in the organizations.

Not surprisingly since almost all CASE-tools contain at least functionality for conceptual modeling (Hewett and Durham, 1989), this functionality is used by almost all users. The percentage of the source code being produced by the code-generation facilities varied between

Table 13. Use of CASE-tools.

Usage area	Percentage of users
Conceptual modeling (ER, DFD etc.)	92%
Drawing of screens and reports	54%
Storing, administration and reporting of system information	54%
Code generation	54%
Prototyping/simulation for validation	46%
Generation of DB-schema	46%
Project and process management	31%
Consistency checking of specifications	23%
System test	15%
Reverse engineering	8%

10 and 100% , with an average of 48%. In half of the organizations using code-generation, later maintenance was performed on the specification and design level for later regeneration of code. In the other half, further maintenance was performed on the generated code.

4.3. Perception on the benefits of CASE-tools

On the perceived benefits of applying CASE-technology, the numbers in Table 14 were reported. A five point scale with 5 indicating *very important* and 1 *not important* was used. The first number indicate the total mean, the second is the mean assessed value from those applying CASE, and the third is the mean assessed value from those not currently applying CASE. Δ is the difference between these two values, and *p* indicates the 2-tail significance value for equality of these means using a non-parametric test.

In Tables 15 and 16, the two groups are split and the tables are sorted on means.

In Tables 17 and 18 the two groups are similarly split according to the percentage of respondents that regarded the aspect as very important.

From the presented tables, we see that both groups seem to agree on the importance of CASE for improved maintainability of application systems, and also the reduction of number of errors in the system, even if the CASE-users are not so optimistic on this. Worth

Table 14. Perceived benefits of CASE.

	Aspect	All	User	No-user	Δ	*p*
a	Increase the productivity of the developer	3.9	3.6	4.0	.3	.3058
b	Support rapid prototyping and validation	3.8	3.9	3.7	−.2	.4929
c	Simplify the development process	3.8	3.8	3.8	−.1	.7657
d	Formalize/standardize the development process	4.2	4.3	4.1	−.2	.4041
e	Reduce the time for application development	3.8	3.5	3.9	.4	.2603
f	Reduce the cost of application development	3.8	3.5	3.9	.3	.3939
g	Improve the interface toward the system	3.0	3.3	2.8	−.5	.2586
h	Integration of development phases and tools	3.2	3.6	2.9	.7	.0921
i	Automatic generation of documentation	3.6	3.6	3.7	.1	.8176
j	Standardization of documentation	3.8	4.1	3.7	−.3	.3153
k	Increased possibility of reuse	3.7	3.8	3.7	−.1	.5530
l	Improved maintainability	4.2	4.2	4.2	0	.8630
m	Automatic code generation	3.3	3.2	3.4	.2	.9149
n	Better control of the application development	3.1	3.8	3.7	−.2	.6921
o	Automating the project management	2.5	2.6	2.5	−.1	.8176
p	Automatic consistency checking	3.2	3.0	3.3	.3	.6780
q	Reduce the number of errors in the system	4.0	3.8	4.1	.3	.3697
r	Fulfill user requirements	3.6	4.1	3.4	.7	.2389
	All over mean	3.626	3.666	3.609		

Table 15. Benefits of using CASE, assessed by CASE-users.

	Aspect	Mean
d	Formalize/standardize the development process	4.3
l	Improved maintainability	4.2
j	Standardization of documentation	4.1
r	Fulfill user requirements	4.1
b	Support rapid prototyping and validation	3.9
c	Simplify the development process	3.8
k	Increased possibility of reuse	3.8
n	Better control of the application development	3.8
q	Reduce the number of errors in the system	3.8
a	Increase the productivity of the developer	3.6
h	Integration of development phases and tools	3.6
i	Automatic generation of documentation	3.6
e	Reduce the time for application development	3.5
f	Reduce the cost of application development	3.5
g	Improve the interface toward the system	3.3
m	Automatic code generation	3.2
p	Automatic consistency checking	3.0
o	Automating the project management	2.6

Table 16. Benefits of using CASE, assessed by non-users.

	Aspect	Mean
l	Improved maintainability	4.2
q	Reduce the number of errors in the system	4.1
a	Increase the productivity of the developer	4.0
d	Formalize/standardize the development process	3.9
e	Reduce the time for application development	3.9
f	Reduce the cost of application development	3.9
c	Simplify the development process	3.8
b	Support rapid prototyping and validation	3.7
i	Automatic generation of documentation	3.7
j	Standardization of documentation	3.7
k	Increased possibility of reuse	3.7
n	Better control of the application development	3.7
r	Fulfill user requirements	3.4
m	Automatic code generation	3.4
p	Automatic consistency checking	3.3
h	Integration of development phases and tools	2.9
g	Improve the interface toward the system	2.8
o	Automating the project management	2.5

Table 17. Reasons being very important for using CASE, assessed by CASE-users.

	Aspect	Percentage
d	Formalize/standardize the development process	50
r	Fulfill user requirements	50
k	Increased possibility of reuse	50
b	Support rapid prototyping and validation	45
j	Standardization of documentation	33
i	Automatic generation of documentation	30
l	Improved maintainability	27
n	Better control of the application development	27
q	Reduce the number of errors in the system	25
g	Improve the interface toward the system	22
h	Integration of development phases and tools	20
c	Simplify the development process	18
f	Reduce the cost of application development	18
o	Automating the project management	10
a	Increase the productivity of the developer	9
e	Reduce the time for application development	9
m	Automatic code generation	9
p	Automatic consistency checking	0

Table 18. Reason being very important for using CASE, assessed by non-users.

	Aspect	Percentage
q	Reduce the number of errors in the system	50
l	Improved maintainability	43
b	Support rapid prototyping and validation	41
r	Fulfill user requirements	37
a	Increase the productivity of the developer	36
f	Reduce the cost of application development	36
d	Formalize/standardize the development process	35
e	Reduce the time for application development	29
j	Standardization of documentation	26
i	Automatic generation of documentation	20
c	Simplify the development process	19
k	Increased possibility of reuse	19
n	Better control of the application development	19
m	Automatic code generation	19
p	Automatic consistency checking	10
g	Improve the interface toward the system	5
o	Automating the project management	5
h	Integration of development phases and tools	0

mentioning though, is that the factor of fulfilling user requirements seems to be more highly regarded as a benefit of CASE among CASE-users, even if this is not statistically significant. Neither is any of the other differences, thus we do not reject hypothesis H8. On the other hand, note the difference on the question of integration of phases and tools, which are one of the areas where integrated CASE tools are supposed to be most useful. The benefit for increased productivity is assessed to be larger among those not using CASE than among those using it. The non-users are also more positive on the effects of reducing time and cost of development than those using it. This is in line with other investigations (Kusters and Wijers, 1993). The benefit of CASE for productivity has generally been a selling point among CASE-vendors whereas investigations have shown that productivity in the short run in fact tends to decrease whereas quality of the produced solutions tends to increase (House, 1993).

To investigate this area further, a factor-analysis (Norusis, 1993) was attempted, but the test for applicability of this analysis gave a KMO-value of 0.4 which highly discourage such analysis. We are also aware of that the technique that was used on this question has certain flaws, since other benefits not in the list will seldom be mentioned, even if it is room for suggesting own categories in the form. In Dekleva (1992a) it is illustrated that other techniques can give different overall results on questions of this kind.

5. Concluding remarks

We have in this article presented some of the results from a survey investigation performed among Norwegian organizations in the area of development and maintenance of information systems regarding development and maintenance methodology and the use of CASE-tools.

Revisiting our hypothesis we conclude the following:

- H1: There is no statistically significant difference between the maintenance effort in organizations that have a comprehensive development methodology and those that don't.
  Not rejected.
- H2: There is no statistically significant difference between the functional maintenance effort in organizations that have a comprehensive development methodology and those that don't.
  Rejected, those organizations having a comprehensive development methodology use significantly less effort on functional maintenance.
- H3: There is no statistically significant difference between the size of the organizations that have a comprehensive development methodology and those that don't.
  Rejected, those organizations having a comprehensive development methodology were significantly larger in most respects.
- H4: There is no statistically significant difference between the maintenance effort in organizations that use CASE-tools in development and maintenance and those that don't.
  Not rejected.
- H5: There is no statistically significant difference between the functional maintenance effort in organizations that use CASE-tools in development and maintenance and those that don't.
  Not rejected.

- H6: There is no statistically significant difference between the size of the organizations that use CASE-tools in development and maintenance and those that don't.
  Rejected, those organizations using CASE-tools were significantly larger in all respects.
- H7: There is no statistically significant difference between the size of organizations when it comes to how they develop their major systems (customized or packaged development).
  Rejected: Organizations with large data departments and many end-users have a larger percentage of systems developed by the data department and a smaller percentage of application systems developed by outside companies. Organizations with small data-departments and few end-users use a larger percentage of packages with small adjustments.
- H8: There is no statistically significant difference in the perception of the benefits of CASE between users and non-users of CASE-tools.
  Not rejected, even if certain interesting differences which should be further investigated can be observed.
- H9: The institutionalization of organizational controls do not influence the behavior during maintenance significantly.
  Rejected in the general case. With some exceptions, there is a clear connection between the instituted organizational control and how maintenance and change request handling is performed.

It appears that larger organizations with larger data departments have a more widespread use of a comprehensive development methodology. It is also primarily those organizations who have started to apply CASE-tools, and they also perform better when it comes to functional maintenance. Thus even if one would expect that smaller organizations could get their work done efficiently using a more loosely formalized development and maintenance methodology, this appears not to be so. On the other hand, having a complete development method does not appear to influence the amount of traditional maintenance significantly. This is in step with the results of Dekleva (1992b) which showed that there is no conclusive evidence that organizations using modern development methods use less time on maintenance activities. On the other hand they spend a larger proportion of the time on functional perfective maintenance, which, other things being equal, decreases the amount of functional maintenance.

When it comes to general conclusion regarding CASE-use, the small number of our respondents applying CASE-tools should make us very cautious in coming with any strong statements on the overall use of CASE-tools in Norway. This was neither the main motivation behind the investigation. Comparing with other investigations though, it seems that the application of CASE-tools in Norway might be less widespread than what is reported from other European countries such as the Netherlands and UK (Hardy et al., 1995; Kusters and Wijers, 1993).

Acknowledgments

This article is an extended version of (Krogstie, 1995c) and we would like to thank the reviewers of the article for their inputs and comments. We would also like to thank all the participants of the survey-investigation for their effort in filling out the forms and everyone helping us in the refinement of the questionnaire. Arne Henne deserves to be mentioned for providing the forms of his survey. Arne Sølvberg initiated the investigation and has supported the work both spiritually and financially.

References

Arnold, R.S. and Parker, D.A. 1982. The dimensions of healthy maintenance. In *Proceedings of the 6th International Conference on Software Engineering (ICSE)*, IEEE Computer Society Press, pp. 10–17.

Bergersen, L. 1990. *Prosjektadministrasjon i Systemutvikling. Aktiviteter i Planlegningsfasen Som Påvirker Suksess.* (In Norwegian), Ph.D. Thesis, ORAL, NTH, Trondheim, Norway.

Central Bureau of Statistics of Norway. 1992. *Statistical Yearbook of Norway.* Statistisk sentralbyrå, Oslo, Norway.

Dekleva, S.M. 1992a. Delphi study of software maintenance problems. In *Proceedings of the Conference on Software Maintenance (CSM'92)*, pp. 10–17.

Dekleva, S.M. 1992b. The influence of the information systems development approach on maintenance. *MIS Quarterly*, pp. 355–372.

Dekleva, S.M. 1992c. Software maintenance: 1990 status. *Journal of Software Maintenance*, 4:233–247.

Hardy, C., Stobart, S., Thompson, B., and Edwards, H. 1995. A comparison of the results of two surveys on software development and the role of CASE in the UK. In *Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering (CASE'95)*, H.A. Müller and R.J. Norman (Eds.), IEEE Computer Society Press, Toronto, Canada, pp. 234–238.

Henne, A. 1992. Information systems maintenance—Problems or opportunities?. In *Proceedings of Norsk Informatikk Konferanse 1992 (NIK'92)*, Tromsø, Norway, pp. 91–104.

Hewett, J. and Durham, T. 1989. CASE: The next steps. Technical report, OVUM.

House, C. 1993. Case studies in the CASE industry. *Keynote Speech CASE'93.*

Krogstie, J. 1994a. Information systems development and maintenance in Norway: A survey investigation. *Norsk Konferanse for Organisasjoners Bruk av Informasjonsteknologi (NOKOBIT'94)*, Bergen, Norway, pp. 1–22.

Krogstie, J. 1994b. Survey investigation: Development and maintenance of information systems in Norway. Technical Report 0802-6394 6/94, IDT, NTH, Trondheim, Norway.

Krogstie, J. 1995a. *Conceptual Modeling for Computerized Information Systems Support in Organizations.* Ph.D. Thesis, IDT, NTH, Trondheim, Norway.

Krogstie, J. 1995b. On the distinction between functional development and functional maintenance. To be published in *Journal of Software Maintenance.*

Krogstie, J. 1995c. Use of development methodology and CASE-tools in Norway: Results from a survey. In *Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering (CASE'95)*, H.A. Müller and R.J. Norman (Eds.), Toronto, Canada, pp. 239–248.

Krogstie, J. and Sølvberg, A. 1994. Software maintenance in Norway: A survey investigation. In *Proceedings of International Conference of Software Maintenance (ICSM'94)*, H.A. Müller and M. Georges (Eds.), IEEE Computer Society Press, pp. 304–313. Received "Best Paper Award".

Kusters, R.J. and Wijers, G.M. 1993. On the practical use of CASE-tools. *Results of a Survey*, in Lee, Reid, and Jarzabek (1993), pp. 2–10.

Lee, H.Y., Reid, T., and Jarzabek, S. (Eds.) 1993. In *Proceedings of the 6th International Workshop on Computer-Aided Software Engineering (CASE'93)*, IEEE Computer Society Press, Singapore.

Lientz, B.P. and Swanson, E.B. 1980. *Software Maintenance Management.* Addison Wesley.

Norusis, M.J. 1992. *SPSS for Windows: Base System User's Guide.* Chicago, Illinois, USA: SPSS Inc.

Norusis, M.J. 1993. *SPSS for Windows: Professional Statistics.* Chicago, Illinois, USA: SPSS Inc.

Nosek, J.T. and Palvia, P. 1990. Software maintenance management: Changes in the last decade. *Journal of Software Maintenance* 2:157–174.

Parkinson, J. 1990. Making CASE work. In *CASE on Trial*, K. Spurr and P. Layzell (Eds.), John Wiley & Sons, pp. 213–242.

Stobart, S.C., Thompson, J.B., and Smith, P. 1991. Use, problems, benefits, and future direction of computer-aided software engineering in the United Kingdom. *Information and Software Technology*, 33(9):629–636.

Stobart, S.C., van Reeken, A.J., Low, G.C., Trienekens, J.J.M., Jenkins, J.O., Thompson, J.B., and Jeffery, D.R. 1993. An empirical evaluation of the use of CASE tools. In Lee et al. (1993), pp. 81–87.

Swanson, E.B. and Beath, C.M. 1989. *Maintaining Information Systems in Organizations*. Wiley Series in Information Systems, John Wiley & Sons.

van Swede, V. and van Vliet, H. 1994. Consistent development: Results of a first empirical study of the relation between project scenario and success. In *Proceedings of the 6th International Conference on Advanced Information Systems Engineering (CAiSE'94)*, G. Wijers, S. Brinkkemper, and T. Wasserman (Eds.), Springer Verlag, Utrecth, Netherlands, pp. 80–93.

Wijers, G.M. and van Dort, H.E. 1990. Experience with the use of CASE-tools in the Netherlands. In *Proceedings of the Second Nordic Conference on Advanced Information Systems Engineering (CAiSE'90)*, B. Steinholtz et al. (Eds.), Lecture Notes in Computer Science, Springer-Verlag, Stockholm, Sweden, 436:5–20.

Automated Software Engineering 3, 369–390 (1996)
© 1996 Kluwer Academic Publishers. Manufactured in The Netherlands.

A Debugging and Testing Tool for Supporting Software Evolution

D. ABRAMSON davida@cit.gu.edu.au
R. SOSIC sosic@cit.gu.edu.au
School of Computing and Information Technology, Griffith University, Kessels Rd, Brisbane, Queensland, 4111

Abstract. This paper describes a tool for debugging programs which develop faults after they have been modified or are ported to other computer systems. The tool enhances the traditional debugging approach by automating the comparison of data structures between two running programs. Using this technique, it is possible to use early versions of a program which are known to operate correctly to generate values for comparison with the new program under development. The tool allows the reference code and the program being developed to execute on different computer systems by using open distributed systems techniques. A data visualisation facility allows the user to view the differences in data structures. By using the data flow of the code, it is possible to locate faulty sections of code rapidly. An evaluation is performed by using three case studies to illustrate the power of the technique.

Keywords: automatic software testing, debugging, testing, evolutionary software

1. Introduction

The high cost of software development, in combination with advances in software engineering, has caused the emergence of a software development methodology based on *evolution*. In this methodology, new programs are built from existing ones, utilising sections of code from old programs to perform functions in the new application. The methodology is most commonly applied when the functionality of a program is expanded incrementally, or when code is *ported* from one system to another. In the latter case, the old code is moved to a new platform with as little modification as possible. However, this simple minded approach often fails. For example, it may be necessary to modify sections of code which are system dependent, and in some circumstances it may even be necessary to rewrite the entire program in another programming language. Further, subtle differences in the semantics of programming languages and operating systems mean that the code may behave differently on two systems. Because of these practical considerations, it is desirable that software tools are available to simplify the process as much as possible.

Traditional debuggers such as *dbx* (Adams and Muchnick, 1986; Linton, 1990; Sun Microsystems, 1990) and *gdb* (Stallman), and others (Moher, 1988; Olsson, 1991; Ramsey, 1992; Satterhwaite, 1972; Cheng and Hood, 1994) do not emphasise the debugging and testing of applications which change during their development cycle. Debuggers of this type allow the user to manipulate the new program through process control commands, and to examine and modify the state of the program. In debugging the program, the user must determine possible modes of failure, and then stop the execution at key points to examine the state. In determining whether the state is correct or not, the user must be able to predict the state values. The prediction is usually based on a detailed knowledge of the operation of the code. This can be extremely difficult for complex code, especially if the person performing the debugging is not the author of the original program. Most importantly, existing debuggers do not try and make use of any other versions of the program in helping the user form sensible predictions for state values.

In this paper we discuss a new debugging tool which can be used to test programs which change over time Our tool incorporates conventional debugger commands as well as a new set of commands which facilitate comparison of the new program with an existing version. The paper begins with a discussion of the current techniques used to test evolutionary programs. It then describes GUARD[1] (Griffith University Relative Debugger), a new debugger developed at Griffith University, followed by some implementation considerations. Finally, we provide an evaluation of the technique using a number of case studies which highlight different aspects of GUARD.

2. How do we test and debug evolving programs?

Figure 1 shows a number of classifications for changes that a program may encounter during its lifetime. It often begins as a small program for testing some basic functionality and design. It may be augmented and incrementally expanded into a large software system, and some of the core algorithms may even be altered. These types of modifications can be attributed to changes in the functionality or operation of the code. In figure 1 changes which alter the core algorithms or augment the program with new ideas are classified as functional changes. The program may be converted for execution on different hardware and software platforms, and may even be re-written in another language to allow it to take advantage of new hardware. These types of modification can be classified as migratory ones. In figure 1 changes attributed to rewriting the program in another language or porting it to another computer system are classified as migratory. Regardless of the cause of the changes, at each of these stages the programmer must determine whether the program is still operating correctly, and if not, must determine which of the alterations caused the new behaviour.

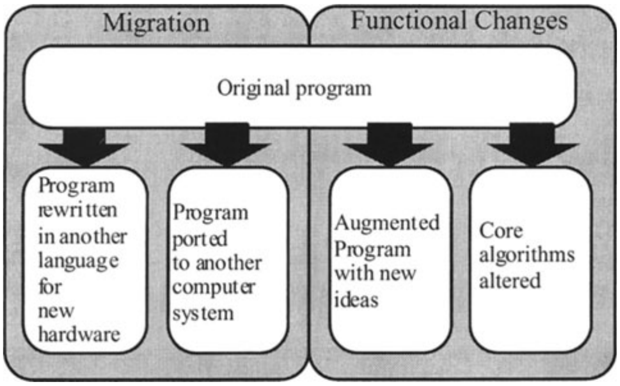

Migration
Functional Changes
Original program
Program rewritten in another language for new hardware
Program ported to another computer system
Augmented Program with new ideas
Core algorithms altered

Figure 1. Classification of program changes.

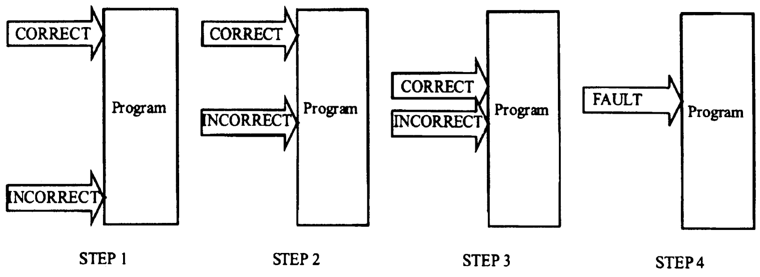


Figure 2. Refining the erroneous region.

Currently, there are very few tools which assist porting and debugging of evolving programs across different hardware and software platforms. Traditional debuggers have a number of severe limitations which significantly reduce their applicability. The most serious limitation is that they are incapable of dealing with several programs running at the same time and possibly on different computers. They have no facilities for comparing the output and internal state of different programs, which must be done by tedious and error prone manual methods.

The most common technique for testing and debugging evolved programs is to use the data flow of the code to locate the point at which the data structures in the new program diverge from those in the existing one. Thus, the existing code acts as a *reference* version by defining a set of expectations. In this way, the user typically works back to determine the last point at which the data was correct by comparing data structures with the reference version. The process is applied iteratively until the faulty region is refined to a small segment of the program. Once this point has been established, most errors can be traced quickly to a small section of faulty code, and the error can be corrected. This technique is illustrated in figure 2, which shows that relatively few stages can be used to refine the faulty region to a manageable size.

Debugging real programs using this technique with currently available tools can be tedious and error prone. Typically, output statements are placed in both the reference and the debugged code, and the values are either compared by visual inspection, or by a file comparison program (Galbreath et al., 1994). If the two programs execute on different computer systems then the data must be transferred before it can be compared.

These shortcomings have motivated the development of a new debugging tool, which is described in the next section.

3. GUARD: A relative debugger

3.1. What is GUARD?

GUARD is a distributed debugger which operates in an open heterogenous computing environment (Abramson and Sosic, 1995; Sosic and Abramson). GUARD provides the user with functionality to control more than one program and to make assertions about the correctness of a new program with reference to an existing one. Consequently, GUARD supports the evolution of programs because it acknowledges the existence of working versions of the

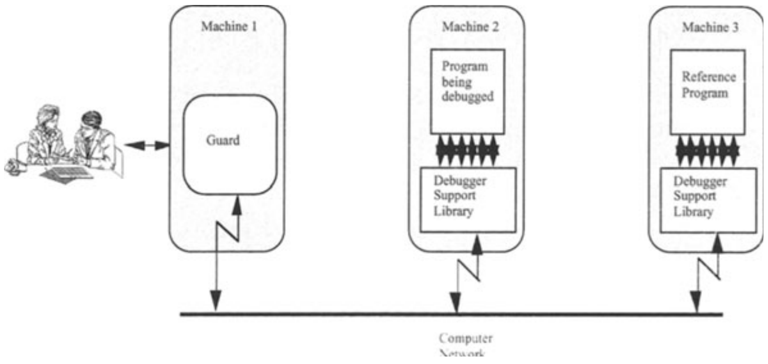


Figure 3. GUARD—A relative debugger.

code. Figure 3 shows the way that GUARD interacts with both the reference code and the code being debugged.

Because GUARD works in a distributed environment, it is possible to execute the reference code, the program being debugged and GUARD itself on three different computer systems, as shown in figure 3. GUARD communicates with the programs it is controlling via a network and makes use of a special debugger support library [called Dynascope (Sosic, 1995)] which interacts with the applications.

GUARD relies on the premise that differences between the contents of the data structures in the reference code and the one being debugged can be used to detect faults. This assumes that the two programs utilise comparable data structures, or at least provide a set of conversion routines which make them appear equivalent. GUARD makes no assumptions about control flow in the two programs, which may be different. It requires the user to determine key points in the two programs at which various data structures should be equivalent. The overall debugging process is as follows. The user determines that the new code is erroneous by observing that key data structures are incorrect after execution. GUARD is then used to test equivalence of these data structures between the reference and debugged programs at various points in the code. By tracing the data structures back to their sources using the dataflow of the program, it is possible to find the latest point at which the two codes are equivalent and the earliest point at which they diverge. The error then lies between these two points. This overall technique is used routinely in many disciplines, such as debugging electronic circuits. In electronics, test equipment is used to compare observed signals with those recorded on working equipment. When applied to debugging computer programs, the process is normally performed manually using very little computer support and it is usually quite laborious. The main power of GUARD is that it provides the facilities to make the process very simple and efficient. It is effective because the user can use a divide-and-conquer technique on very large systems and reduce the suspect section of code to a manageable size.

GUARD complements conventional debuggers, rather than replacing them. By allowing the user to determine quickly whether two programs compute the same values it is possible to find out where the two codes diverge. Once the point of divergence has been located, conventional debugging techniques can be used to isolate the faulty code. For this reason, the core GUARD functionality may be embedded in a conventional debugger.

The availability of other supporting tools can dramatically improve the effectiveness of this overall technique. For example, if the data flow of the code is available during the debug and test phase, then this can be used to assist the user in determining where to test the new code against the reference. Consequently, GUARD could be embedded in a CASE environment which gives concurrent access to the source, various call tree graphs and dataflow representations of the program.

3.2. *Network independent debugging*

GUARD makes use of open distributed processing techniques to allow the reference code and the debugged code to execute concurrently on different computer systems. This means that when a program is being ported to another system, the reference code operates in an environment which is known to produce the correct results. Consequently, the user is able to concentrate on the causes of deviation of the code to be debugged. Network independence and support for heterogenous computing platforms make heavy demands on the underlying technology, and some of these issues will be discussed in Section 4.

Network location information is restricted to the GUARD command `invoke`, which starts executing a process. In this command a logical name is bound to the process for use in future commands. After the `invoke` command has been issued, all other GUARD commands use the logical name when referring to the process. GUARD hides the details of the underlying architecture from the user, and thus it is possible to compare data structure contents across widely different machines. For example, one machine may represent integers using 64 bit big endian addressed words, and the other may use 32 bit little endian addressed words. GUARD automatically translates the external data into a generic internal form before it performs the comparison. We have tested GUARD across a wide range of Unix platforms using the Internet as the underlying network. This feature of GUARD will be illustrated by one of the case studies reported in this paper.

3.3. *Using GUARD*

GUARD relies on the user to make a number of assertions which compare data structures in the code to be debugged and the reference version. These assertions make it possible to detect faulty code because they indicate where (and when) the data structures deviate from those in the reference code.

The choice of data structures and test points must be determined by the user based on some knowledge of the application. It is not necessary to test all data structures, but only those which will help uncover the source of the error. Likewise, it is not necessary to test the structures after each executable statement. A search which refines the faulty region, such as a binary search, can be very effective.

GUARD can be used in two modes: one in which the assertions are specified *declaratively*, and the other using *procedural* techniques. Both techniques do not require any recompilation of the code and make use of debugger breakpoints to interrupt the code being tested. The procedural scheme relies on the user manually placing breakpoints in both the reference code and the debugged code. These are planted by the user at key test points. Both programs are then executed until they reach these breakpoints, after which the user tests the equality of arbitrary data structures using a `compare` statement. The following example shows the syntax of the *compare* statement by comparing the values of variable *test* in the reference code and the debug code. In this example, *reference* and *debug* are names which are bound to the two processes under examination by the `invoke` command.

```
compare reference::test = debug::test
```

If the two data structures are not equivalent, then the nature of the error is reported. If the variables are simple types like integer or real, then it is possible to report the two values. If they are compound structures like arrays, then the location of the difference in the two structures must be reported as well. Later in the paper we will describe the method for reporting differences in array structures. After the comparison, the user can resume execution of the two programs using a `continue` statement. It is possible to compare a number of data structures after the breakpoints have been reached. New breakpoints can also be set to further refine the erroneous code at any stage of the debugging process. This process closely resembles the way a conventional debugger is used. However, it allows the user to control two processes concurrently and compare their data structures. This manual scheme can become unwieldy on large programs because there are two processes to control. Further, it is not well-suited to using the debugger to automatically test whether a new version of program matches a previous one, because it requires a great deal of user interaction. Consequently, we have developed an automatic mode of operation called *declarative assertions.*

Declarative assertions allow the user to bind a set of breakpoints to comparison operations at any time prior to, or during, execution of the code. In this way, the comparisons are invoked automatically each time the breakpoints are reached. If the compare statements do not detect an error, then the code is automatically resumed without any user interaction. Declarative assertions provide a convenient mechanism for stating a set of conditions for correct execution, and thus are well suited for automatically testing a new program against previous versions. If the assertions do not cause any errors, then the code is assumed to conform to previous versions. Declarative assertions are also effective when an error is detected only after a number of iterations of a particular statement. Because the user is not involved until an error is detected, little user interaction is required to actually detect the erroneous lines. The following syntax is used to declare an assertion:

```
assert reference::variable1@line1 = debug::variable2@line2
```

where `reference` and `debug` are bound to the two processes as discussed previously, `variable1` and `variable2` are arbitrary variables in the programs, and `line1` and `line2` are source code line numbers. In Section 5 we will illustrate the use of declarative assertions for detecting errors in a large scientific modelling program.

3.4. Data types

Like conventional debuggers, GUARD needs to understand the types of variables in the programs it controls. On many systems (e.g., Unix) the type information is embedded in special symbol table information stored in the executable image. This can be automatically extracted by the debugger at the beginning of a debug session. In a conventional debugger, this information allows it to display variables in a way which is meaningful to the programmer. In GUARD, the information is also required so that it knows how to compare variables. For example, variables must be type compatible before they can be compared. Further, GUARD needs to understand the structure of the variables during the comparison operation so that it can traverse the basic elements. Other complications arise because the reference code may execute on a system with different attributes from the code being tested. For example, byte ordering may differ and the two programs may even be written in different languages. Consequently, GUARD must map types from one system onto another.

GUARD currently handles variables of base types integer, real, character and multi dimensional arrays of these base types. The base types are compared for equality. Arrays are compared by traversing their elements in order. Differences are reported together with their position information. This allows GUARD to compare arrays in programming languages which use different ordering, such as Fortran and C. GUARD allows also comparisons of sub-arrays.

These base types have been sufficient to demonstrate the effectiveness of GUARD in debugging a number of large scientific programs. In future versions of GUARD we will add support for data types such as records and dynamic structures like linked lists. Records can be handled by walking through the elements of the record recursively until a base type is found. Linked lists require more sophisticated procedures for mapping the data structures into an intermediate form prior to comparison and for interpretation of this intermediate form. It would also be possible to compare user defined types in an object oriented environment by allowing GUARD to call the object access methods of the appropriate class.

3.5. Tolerances

A program may not be in error simply because its variables do not exactly match those of a reference version. The most obvious example is when the two systems use different floating point representations or libraries. In this case numbers may be deemed to be equivalent if they are within a predefined tolerance of each other. Accordingly, GUARD includes a user controlled tolerance value, below which numbers are considered equivalent. Further, individual assertions may specify their own tolerance value rather than using the global one.

We have experimented with two different types of tolerance, one absolute and the other relative. When absolute tolerances are used, the magnitude of the difference between the reference and the test variables is compared to the tolerance value. When relative tolerances are used, the difference is divided by the larger of the two variables. The latter is required when the numbers are quite small, because even a small absolute difference may constitute a large variation. User interaction is required in order to determine which type of tolerance to use.

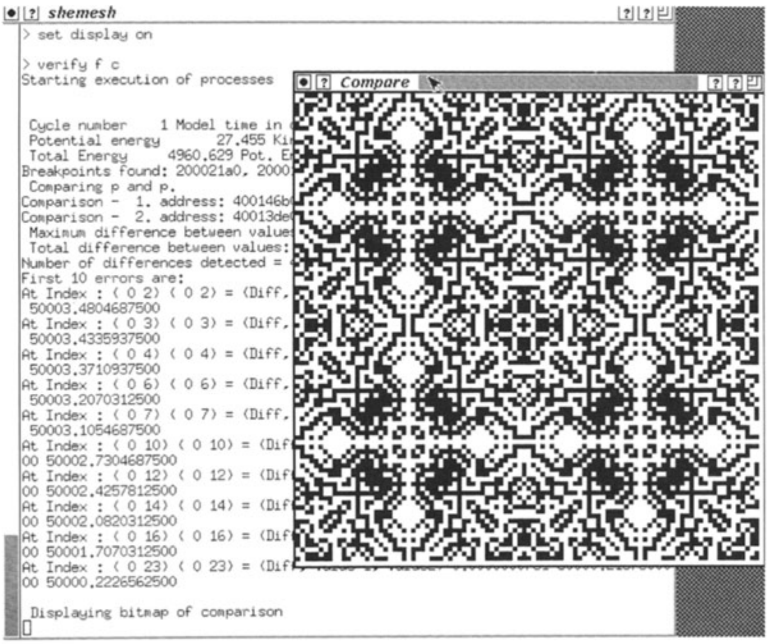
3.6. Displaying the results

If two scalar variables differ then it is possible to display the two values and report the difference. However, when complex data structures differ, it is difficult to interpret the nature of the difference by viewing the numeric values alone, particularly if they are floating point numbers. Consequently, we have developed a simple visualisation system for GUARD which uses a pixel map to show array differences. GUARD also reports the maximum and average differences between arrays as numeric values.

The most intuitive display is formed when two dimensional arrays are visualised. In this case, a two dimensional pixel map is created in which each pixel corresponds to one array element. Errors which are caused by incorrect loop bounds and strides are displayed as regular patterns, making them easy to detect. GUARD currently maps all other shaped arrays onto two dimensional ones by either expanding the one dimensional arrays or merging higher dimensions.

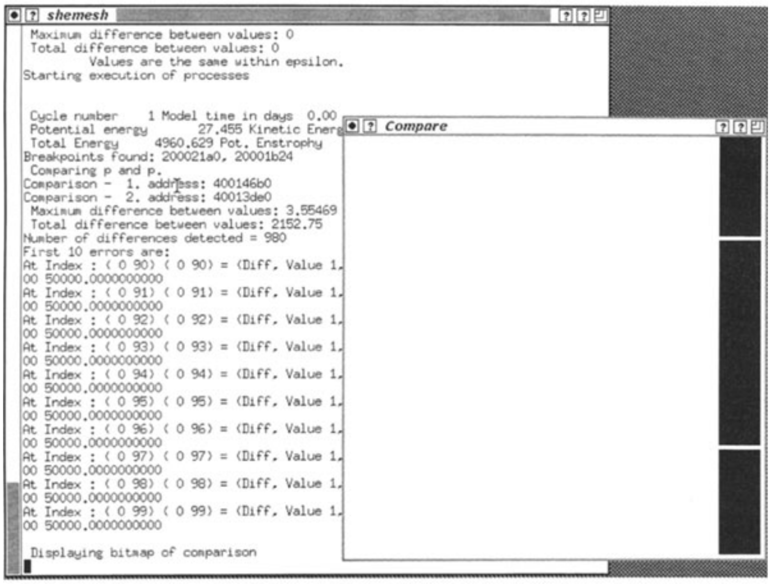
Figures 4(a) and 4(b) show some sample visualisations which were produced by GUARD when it was used for testing a new version of the Shallow Water Equations (Abramson, Dix, and Whiting, 1991). The original code was written in FORTRAN and the new version was written in C and ran on a different computer system. Figure 4a shows the effect of floating point differences between the two dimensional data structures used to hold the *pressure* of the wave. In both visualisations, a black pixel means that the data is different at the specified row and column of the arrays. From figure 4a it can be seen that the two data structures are similar but not exactly the same because many of the values are incorrect, but the maximum difference is quite small (4.3×10^{-06}). Moreover, the maximum difference increases in time as the two programs diverge further. Figure 4b shows the effect of a wrong loop bound in the new code. The section of the array which has not been calculated can be clearly seen as a band of black pixels covering the missing columns on the right.

In figure 5(a) we illustrate a more powerful visualisation of differences. In this case, we show a three dimensional error iso-surface of a particular data structure. The example comes from the comparison of two different weather models, and the displays show an error iso-surface where the error exceeds 0.1% between the temperature variable in the two models (Abramson et al., 1995). Such images convey powerful debugging information to the programmer. For example, by rotating this image in three dimensions it is possible to note that some of the differences are present in the upper layers of the atmosphere, and some are present in the lower levels. This helps to isolate the sections of code which could be causing the divergence, because different pieces of code are responsible for some of the processes which occur in the upper and lower levels of the atmosphere. Also, since there is significant structure to the error surface, it is unlikely to be caused by simple floating point divergence through rounding differences. In this example, there were multiple separable differences, and these are superimposed on the one image. The image in figure 5(b) shows the result of removing one source of errors, as identified using GUARD. It is notable that some errors are still present in the second image. These images were produced using a commercial visualisation package (IBM's Data Explorer). The data is extracted by providing a file name to the assert command, and the data is dumped to the file each time the assertion exceeds the tolerance. A subsequent program processes the data file and imports the data into the visualisation system.



```
> set display on

> verify f c
Starting execution of processes


 Cycle number     1 Model time in
 Potential energy        27.455 Ki
 Total Energy     4960.629 Pot. E
Breakpoints found: 200021a0, 2000
 Comparing p and p.
Comparison -  1. address: 400146b
Comparison -  2. address: 40013de
 Maximum difference between value
 Total difference between values:
Number of differences detected =
First 10 errors are:
At Index : ( 0 2) ( 0 2) = (Diff,
 50003.4804687500
At Index : ( 0 3) ( 0 3) = (Diff,
 50003.4335937500
At Index : ( 0 4) ( 0 4) = (Diff,
 50003.3710937500
At Index : ( 0 6) ( 0 6) = (Diff,
 50003.2070312500
At Index : ( 0 7) ( 0 7) = (Diff,
 50003.1054687500
At Index : ( 0 10) ( 0 10) = (Dif
00 50002.7304687500
At Index : ( 0 12) ( 0 12) = (Dif
00 50002.4257812500
At Index : ( 0 14) ( 0 14) = (Dif
00 50002.0820312500
At Index : ( 0 16) ( 0 16) = (Dif
00 50001.7070312500
At Index : ( 0 23) ( 0 23) = (Dif
00 50000.2226562500

 Displaying bitmap of comparison
```

(a)



```
 Maximum difference between values: 0
 Total difference between values: 0
         Values are the same within epsilon.
Starting execution of processes


 Cycle number     1 Model time in days  0.00
 Potential energy        27.455 Kinetic Energ
 Total Energy     4960.629 Pot. Enstrophy
Breakpoints found: 200021a0, 20001b24
 Comparing p and p.
Comparison -  1. address: 400146b0
Comparison -  2. address: 40013de0
 Maximum difference between values: 3.55469
 Total difference between values: 2152.75
Number of differences detected = 980
First 10 errors are:
At Index : ( 0 90) ( 0 90) = (Diff, Value 1,
00 50000.0000000000
At Index : ( 0 91) ( 0 91) = (Diff, Value 1,
00 50000.0000000000
At Index : ( 0 92) ( 0 92) = (Diff, Value 1,
00 50000.0000000000
At Index : ( 0 93) ( 0 93) = (Diff, Value 1,
00 50000.0000000000
At Index : ( 0 94) ( 0 94) = (Diff, Value 1,
00 50000.0000000000
At Index : ( 0 95) ( 0 95) = (Diff, Value 1,
00 50000.0000000000
At Index : ( 0 96) ( 0 96) = (Diff, Value 1,
00 50000.0000000000
At Index : ( 0 97) ( 0 97) = (Diff, Value 1,
00 50000.0000000000
At Index : ( 0 98) ( 0 98) = (Diff, Value 1,
00 50000.0000000000
At Index : ( 0 99) ( 0 99) = (Diff, Value 1,
00 50000.0000000000

 Displaying bitmap of comparison
```

(b)

Figure 4. (a) Numeric instability causing errors. (b) Incorrect loop bound.

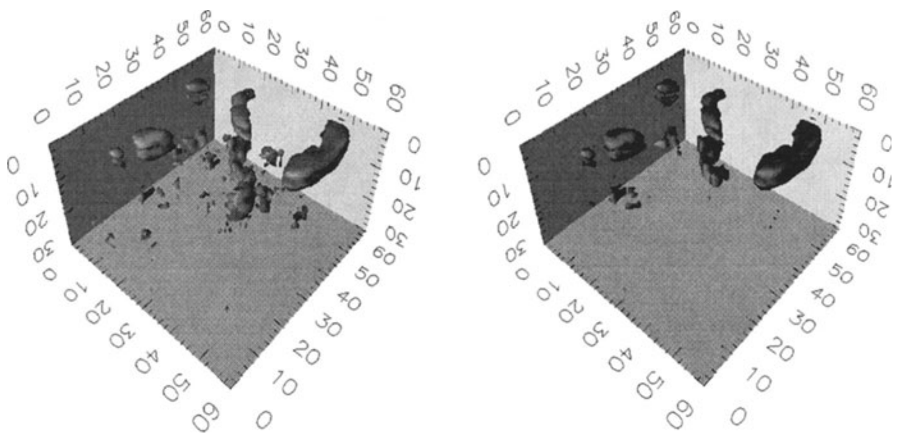


Figure 5. (a) Visualisation of differences. (b) Removal of one source of error.

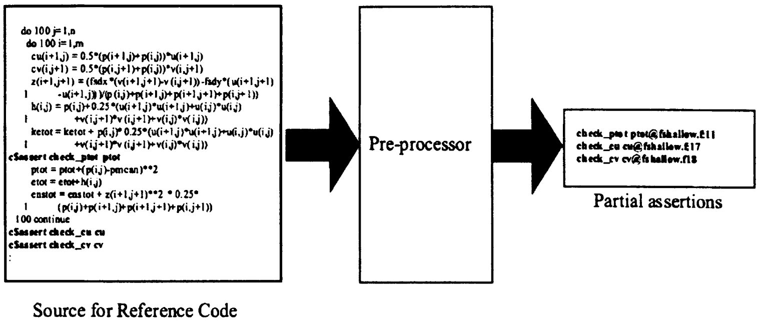
3.7. *Partial assertions*

Assertions contain the names of data structures to compare and the line numbers in the sources where they should be equivalent. In order to create assertions the user must consult the code to determine the correct line numbers. This approach can be error prone when the source changes frequently, because the line numbers also change. Accordingly, we have implemented an additional way of specifying assertions which does not require the user to enter line numbers. In this case, the user writes a *partial* assertion for each of the programs, naming the data structure and an assertion name, and then embeds these in the source files as comments. Partial assertions are extracted from the programs automatically, using a filter program which constructs a file containing assertion names with their corresponding data structure names and line numbers.

GUARD has a special command called `build`, which takes two files with partial assertions and builds a set of complete assertions. If the assertion name appears in both lists then it is matched to form a complete assertion, containing two sets of data structures and associated line numbers. The user is also free to add any additional assertions using the `assert` command. Figure 6 shows how a partial assertion list can be generated. The assertions appear as comments in the code, and thus the program does not need to be recompiled in order to run the program without using the debugger. The partial assertion list contains the name of each assertion and its corresponding variable and line number information. The partial lists are then merged by GUARD into a set of complete assertions.

3.8. *Trace files*

The discussion to date has assumed that the user wishes to execute both the reference code and the program being tested each time the assertions are to be evaluated. However, this



```
   do 100 j=1,n
    do 100 i=1,m
     cu(i+1,j) = 0.5*(p(i+1,j)+p(i,j))*u(i+1,j)
     cv(i,j+1) = 0.5*(p(i,j+1)+p(i,j))*v(i,j+1)
     z(i+1,j+1) = (fsdx *(v(i+1,j+1)-v (i,j+1)) -fsdy*( u(i+1,j+1)
   1        -u(i+1,j)) )/(p (i,j)+p(i+1,j)+p(i+1,j+1)+p(i,j+1))
     h(i,j) = p(i,j)+0.25*(u(i+1,j)*u(i+1,j)+u(i,j)*u(i,j)
   1              +v(i,j+1)*v (i,j+1)+v(i,j)*v( i,j))
     ketot = ketot + p(i,j)* 0.25*(u(i+1,j)*u(i+1,j)+u(i,j)*u(i,j)
   1              +v(i,j+1)*v (i,j+1)+v(i,j)*v( i,j))
c$assert check_ptot ptot
     ptot = ptot+(p(i,j)-pmcan)**2
     etot = etot+h(i,j)
     enstot = enstot + z(i+1,j+1)**2 * 0.25*
   1        (p(i,j)+p(i+1,j)+p(i+1,j+1)+p(i,j+1))
  100 continue
c$assert check_cu cu
c$assert check_cv cv
:
```

```
check_ptot ptot@fshallow.f11
check_cu cu@fshallow.f17
check_cv cv@fshallow.f18
```

Figure 6. Generation of assertions from source code.

is not always convenient. First, the reference program may be executed on a much slower system than the new version, thus evaluating the assertions may take a long time. For example, the new program may be run on a supercomputer. Second, it may not be possible to run the reference code on the original hardware platform, because it may no longer be available. These problems can be solved by storing sufficient information when assertions are first evaluated, so they can be re-executed without running both the application programs. Instead, it is only necessary to re-run the program under test. This technique effectively caches the contents of the data structures for later re-use.

GUARD implements this caching mechanism through a process called *ghost* execution. Ghost execution is performed in two phases. First, the reference code is executed without the new code together with all the necessary assertions. In this phase, the contents of variables being traced are dumped to a file. During the second phase, this file is used as the source of variable values rather that a real reference program being executed. After the user requests ghost execution the debugging can proceed as though the reference code were actually being executed. Providing the user does not specify any assertions which contain variables that have not been cached, the reference code does not been to be re-executed.

3.9. Data extraction and permutation

The discussion to date has assumed that the data structures being compared in the reference and debugged code are identical. Often when code is ported from one system to another there are subtle changes in the data which make this assumption unrealistic. In the case of arrays, it is often necessary to alter the dimensions in the new code to implement additional functionality, or to alter the order of the indexes. Accordingly, GUARD implements array extraction operators and index permutation functions which make it possible to map one array structure to another.

Array extraction is performed using rectangular sub arrays. For example, the description A[5..19][5..19] describes a 15 × 15 sub-array of A starting at row 5 and column 5. This sub-array can be compared with another sub-array providing the sizes are conformant. For example, the following assertion is possible:

```
Assert p1::A[5..19][5..19]@C.c:11 = P2::B[4..18][6..20]@F.f:15
```

Further, changes in index order mean that the new array may be equivalent except that it has a different shape. This permutation is often performed to optimise a program for a new architecture in which the use of vector hardware or cache memory dictates that certain indexes be scanned as inner loops. Accordingly, GUARD implements an arbitrary **permute** function on every assertion, which makes it possible to map index values from one array to another. The following example compares array A with array B using a permutation function which maps index 0 of array A onto index 1 of array B, and index 1 of array A onto index 0 of array B.

```
Assert p1::A@C.c:11 = P2::B@C.c:15/permute (0:1,1:0)
```

Both of these types of assertion were used in comparing the two weather models discussed the third case study in Section 5.3. The new model contained extra rows and columns to allow for interprocessor communication in a parallel form of the program, however, the core data was the same. Also, its row and column ordering was optimised to improve the cache performance.

3.10. Forcing equivalence

Each time an assertion detects a difference in the data between the two programs it reports the divergence to the user and restarts the processes. An option to the assert command instructs GUARD to also copy the data from the reference code data structure into the one in the program being tested, thereby forcing them to contain the same information. This feature has enormous benefits when the user is trying to determine whether the error which has been detected is responsible for some other divergence later in the execution. In one of the case studies in this paper we illustrate the power of the force option when small errors accumulate into larger significant ones after the program has run for some time.

4. Implementation issues

The functionality discussed in the previous sections raises many implementation issues. In this section we briefly touch on some of these, but a more complete discussion is found in (Sosic and Abramson; Abramson, Sosic, and Watson).

4.1. Debugger structure

GUARD is built as a user client which attaches itself to a number of debugger servers. The user client contains all of the user interface code together with the code to manage multiple processes and handle assertions. The client/server structure makes it possible to run the user interface and control logic on one processor and the reference and debugged code on different systems. GUARD is isolated from the implementation details of debug servers by a novel debug library called Dynascope (Sosic, 1995). Dynascope provides functionality which makes it possible to control programs through breakpoints, and to allow extraction of data from variables. A generic interface is provided regardless of the target platform, thus GUARD is portable to the platforms which are supported by Dynascope. Dynascope is currently available on SUN Sparcstations, Next, DEC Alpha, SGI Indy, and IBM RS6000 machines.

4.2. Assertions, event firing and control

Assertions make use of the breakpoint capability which is inherent in most debuggers. However, the logic is more complex than for simple breakpoint handling. When assertions are processed by GUARD, the information relating to the data structure and breakpoint information is stored in an assertion structure. This includes an exact description of the process identifiers, data structures to be compared and the breakpoint addresses.

Subsequently, breakpoints are placed in the two programs at the appropriate places. Then the programs are executed and GUARD waits for breakpoints to be reported. Each time a breakpoint is encountered, the appropriate data structure is extracted from the program and stored in temporary debugger variables, and the program restarted. When both data structures in an assertion are available, the comparison can be performed. This simple event management technique allows the programs to encounter the breakpoints in any order. GUARD performs the comparison only when both breakpoints for an assertion are encountered. If a process encounters another breakpoint at the same address before the previously stored data has been compared, then the data is held in a *first-in-first-out* queue. This preserves the temporal ordering of the data.

Allowing each program to continue execution immediately means that the programs can follow different control structures, at the cost of more complex resource management within the debugger. Data must be retrieved from the user program and saved until it is required, and sufficient space must be available for multiple data structures. At present, we impose a limit on the number of outstanding items for every assertion. If the limit is exceeded the application program is blocked until the assertions are evaluated. Providing the other process encounters a breakpoint in the meantime, the assertion will execute and the two processes can be restarted. However, it is possible to dead-lock a process by limiting the number of items too severely given a particular set of assertions. In our experience, this has not proved to be a limitation[2].

4.3. Issues in heterogeneous distributed computing

Heterogeneous computing platforms pose some interesting challenges for a relative debugger, which must possess the following characteristics:

- The debugger must support the execution of more than one program concurrently;
- The debugger must inter-operate with different platforms;
- The debugger must perform all the necessary data type conversions between platforms and language environments in order to perform meaningful comparisons.

As discussed previously, GUARD uses a debug server, called Dynascope (Sosic, 1995; Sosic, 1995), which hides most of the issues related to the exact process for starting and running programs on distributed platforms. The `invoke` command contains sufficient information to inform Dynascope of the process location, after which there is no need to re-issue this information. Dynascope uses remote execution commands to start remote debug servers, and data is transported using Unix sockets (Stevens, 1990). Dynascope also contains mechanisms to manage heterogenous representations of the same data, including differences in byte ordering, character sets, data sizes and floating point representations. Data from remote systems is mapped into a generic representation before it is returned to the debugger. In this way it is possible to compare data between widely different architectures, and this information can be hidden from the debugger itself. More details of Dynascope can be found in (Sosic, 1995).

5. Evaluation

In this section we evaluate GUARD by considering three case studies, involving real world applications of the technology. Each case highlights a different aspect of relative debugging. In the first case study, GUARD is used to track a difference in a scientific modelling code which occurred as a result of software maintenance. In this case the error is traced to one erroneous source statement in the new version of the code. In the second case study GUARD is used to determine why the same source program behaves differently on two different computer systems. In this example the error is isolated to different behaviour of a mathematical library function. In the final case study, GUARD is used to compare the execution of two different models which should compute the same results. In this case one of the models is a version which has been modified for parallel execution, and is substantially different from the sequential version. In this case, GUARD helps isolate two independent differences in the two models.

5.1. Case study 1—Finding a source error

In this section we describe an application of GUARD for finding a subtle error in a scientific code. The program, a photo chemical pollution code, models the chemical processes which occur during smog formation (McRae et al., 1992). It has been used as part of a number of real world studies involved with formulating pollution control strategies. This software has characteristics which are typical of many other scientific modelling programs. It is written in Fortran and spans 15,000 lines of code over 15 source modules.

One of the key data structures in the program is an array, named C, which holds the *concentrations* of all chemical species for each of the cells in three dimensional space. The array is conceptually indexed by two co-ordinate indices (column number in 2D space and vertical level) and a chemical species number. Whilst this would normally require a three dimensional array, it is actually represented as a one dimensional array. This assists with the vectorisation of the program, and also with dynamic memory allocation. However, it also makes the code harder to understand and debug.

In this study we show how GUARD was used to track an error in the numeric integration code. The problem was detected after the program failed to generate correct results for a particular simulation. A reference version of the code was established and used for the comparison of key data structures. Whilst both versions were run on the same computer, it would have been possible to execute them on different networked systems.

Figure 7 shows the basic computational structure of the program at the outer level. After initialising the key data structures the program enters a loop in which the concentrations of each of the chemical species are calculated at discrete time steps. The program works on sections of the concentration vector (`C`) corresponding to each column of the 3 dimensional space. Once the new concentrations have been computed horizontal transport is performed in 2 directions. The program uses a set of hourly wind vector values a number of times before reading in a new set of vectors. These operations are mostly performed in the source file `airshed.F`.

The assertions shown in figure 8 were used to determine the point at which the concentration vector (`C`) became corrupt. As discussed earlier, the assertions were placed at strategic places in the code to try and locate where the `C` vector became corrupt. These assertions detect that the vector `C` was incorrect at lines 2071 and 1931 but still correct at line 1906. Consequently, the error must be contained in the routine `COLLOOP`.

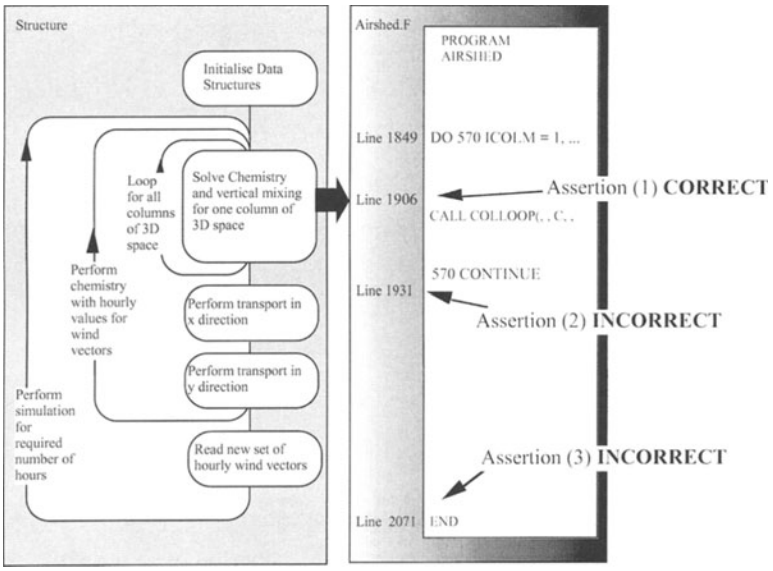

Structure
Initialise Data Structures
Loop for all columns of 3D space
Solve Chemistry and vertical mixing for one column of 3D space
Perform chemistry with hourly values for wind vectors
Perform transport in x direction
Perform transport in y direction
Perform simulation for required number of hours
Read new set of hourly wind vectors
Airshed.F
PROGRAM AIRSHED
Line 1849
DO 570 ICOLM = 1, ...
Line 1906
Assertion (1) CORRECT
CALL COLLOOP(..C..
570 CONTINUE
Line 1931
Assertion (2) INCORRECT
Assertion (3) INCORRECT
Line 2071
END

Figure 7. Overall structure of code plus source in `Airshed.F`.

```
assert old::C@airshed.F:1906 =     new::C@airshed.F:1906     Assertion(1)
assert old::C@airshed.F:1931 =     new::C@airshed.F:1931     Assertion(2)
assert old::C@airshed.F:2071 =     new::C@airshed.F:2071     Assertion(3)
```

Figure 8. Assertions relating to `Airshed.F`.

Figure 9 shows the structure and source of the routine COLLOOP. The appropriate cells from C are copied to a temporary concentration vector called CNT. The new concentrations for the species are calculated by solving the ordinary differential equations which govern the rates of production of each chemical species. Then the concentrations are adjusted to take account of vertical mixing in the column. These two operations are mostly performed in colloop.F. The assertions shown in figure 10 were used to conclude that the values in CNT were correct at line 305 and incorrect at line 321. Consequently, the error must be contained in the routine INTEGR2.

Figure 11 shows the structure and code of the routine INTEGR2, which performs a numeric integration. This makes use of a number of working vectors (such as C3). Figure 12 shows the assertions related to INTEGR2. These assertions determine that C3 was incorrect at 711 and 595, but the switch variable IS was correct at line 587. From the information gathered by these assertions the error was found at line 587 of numerics.F as shown in figure 13.

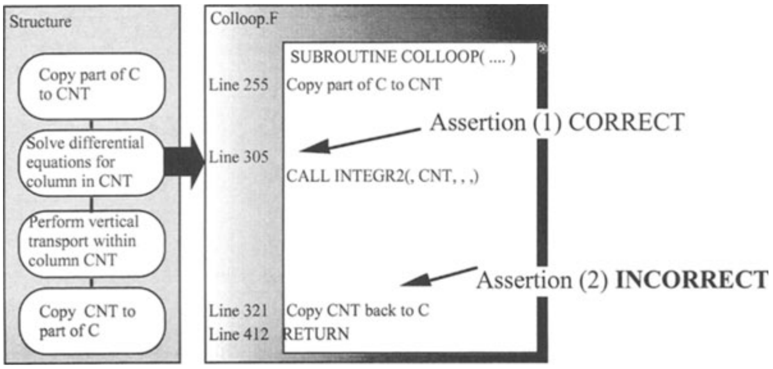

Structure
Copy part of C to CNT
Solve differential equations for column in CNT
Perform vertical transport within column CNT
Copy CNT to part of C
Colloop.F
SUBROUTINE COLLOOP(....)
Line 255 Copy part of C to CNT
Assertion (1) CORRECT
Line 305
CALL INTEGR2(, CNT, , ,)
Assertion (2) INCORRECT
Line 321 Copy CNT back to C
Line 412 RETURN

Figure 9. Structure and code of COLLOOP in Colloop.F.

```
assert old::CNT•colloop.F:305 =    new::CNT•colloop.F:305    Assertion(1)
assert old::CNT•colloop.F:321 =    new::CNT•colloop.F:321    Assertion(2)
```

Figure 10. Assertions relating to Colloop.F.

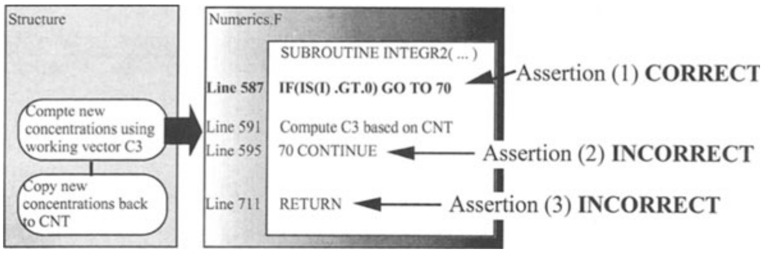

Structure
Compte new concentrations using working vector C3
Copy new concentrations back to CNT
Numerics.F
SUBROUTINE INTEGR2(...)
Line 587 IF(IS(I) .GT.0) GO TO 70
Assertion (1) CORRECT
Line 591 Compute C3 based on CNT
Line 595 70 CONTINUE
Assertion (2) INCORRECT
Line 711 RETURN
Assertion (3) INCORRECT

Figure 11. Structure and code of INTEGR2 in Numerics.F.

```
assert  old::IS●numerics.F:587 =  new::IS●numerics.F:587  Assertion(1)
assert  old::C3●numerics.F:595 =  new::C3●numerics.F:595  Assertion(2)
assert  old::C3●numerics.F:711 =  new::C3●numerics.F:711  Assertion(3)
```

Figure 12. Assertions relating to `Numerics.F`.

`IF (IS(I).GT.0) GO TO 70`	`IF (IS(I).LT.0) GO TO 70`
Correct	Incorrect

Figure 13. Correct and incorrect source in `numerics.F`.

This error caused only some of the array elements to be incorrect, and would have been extremely difficult to trace using a conventional debugger. The difference visualisations allowed the error to be detected very quickly using a simple search in combination with the data flow in the code. One of the main attractions of GUARD was that it was not necessary to alter the code during the debugging phase, and thus new assertions could be developed and refined without the need to recompile the code.

5.2. *Case study 2—Finding errors across platforms*

In this case study GUARD was used to locate the source of a divergence in the pollution code described in the previous section, which occurred after it had been ported from a DEC Alpha workstation to a SUN Sparc Station 5. These two platforms (and their associated operating systems and compilers) differ in a number of important respects. They use different architectures and byte ordering, different default floating point options and different sizes of integers and addresses. Since the source code for the pollution model was identical on the two machines, the divergence could have been due to a variation in any of the hardware, operating system, compilers or run time libraries. Location of this type of problem is a daunting task.

The exact nature of the error can be seen in the error surface plotted in figure 14. This shows where the contents of the concentration array exceeds a 10% relative error tolerance value in the three dimensions of the model. The error appears to be distributed vertically through the atmosphere, which would suggest that the vertical advection code in the model actually transports the error vertically throughout the atmosphere. Also, the error is not present in half of the data structure (the front region as displayed in figure 14), which happens to correspond to a region of space which is above water rather than land. This would suggest that the error is only propagated by some of the physics code relating to pollution transport above land. Finally, the random nature of the error surface suggests that the fault was not caused by a simple array indexing error. More importantly, by viewing the progress of the error surface after each time step of the program, the error can be seen to grow after each iteration of the algorithm. This suggests that the original source of error may be actually be very small, and that it is then magnified by the subsequent computations.

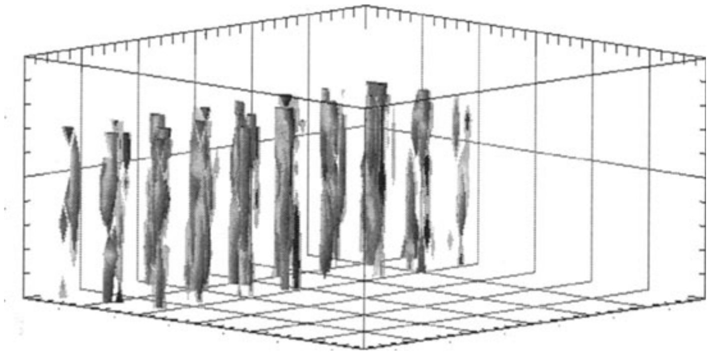


Figure 14. Error surface in pollution model after 60 time steps.

The same technique of divide-and-conquer was used as on the last study. The program was divided a number of times and the key data structures were examined. By using the dataflow of the code it was possible to track a divergence down to a call to the library function EXP. GUARD showed that EXP was returning a result which differed only in the bottom but of the mantissa for some values of operand. However, the error observed after a complete simulation was in the order of 40%. Accordingly, it was not clear that this final error was a result of the small discrepancy detected by GUARD. In order to prove the connection, we used the `force` option discussed in Section 3.10. This option makes it possible to instruct GUARD to force the two programs to use the same concentration array contents whenever a divergence is detected, by copying the data from one program to the other. After performing this operation, the programs produced identical output down to the last binary digit after a complete simulation. Thus, we were able to conclude that the error observed after a complete simulation was caused by an accumulation of very small errors, which happened to be introduced by a different algorithm for EXP. GUARD was able to highlight a much more serious problem with the code, that it was exhibiting chaotic behaviour in the light of very small errors.

This experiment highlights the power of being able to detect differences, and then to force the two programs to use the same data, and continue execution. Without this feature it would have been necessary to write an EXP function which behaved the same on the two systems, and debugging would have taken much longer. GUARD was particularly valuable in this context because it was possible to execute the two programs on the hardware on which the error could be exhibited, and the underlying differences in the platforms could be ignored by the user.

5.3. Case study 3—Finding multiple errors

In the previous two case studies the program being debugged was almost identical to the reference version. In this case study we used GUARD to track a divergence in a large weather model, which was rewritten so that it could be run on a parallel supercomputer. Unlike the previous two examples, this model, MPMM (Foster and Michalakes, 1993; Michalakes et al., 1994), required substantial changes in order to take advantage of the underlying hardware. The original version, MM5 (Anthes, 1986; Grell et al., 1994), was written for sequential vector supercomputers. MPMM had changes in some of the underlying mathematical methods which were better suited to parallel execution. Also, some of the inner-loops in MM5 were promoted to outer loops in MPMM, which appears as a significant change to the source. Further, the order of the indexes in the arrays of MPMM were reversed to improve the performance of the program. In spite of the differences, MM5 and MPMM are supposed to compute the same output. However, one of the output variables of MPMM, the air temperature, was seen to drift from the reference code over some number of time steps. Figure 15 shows an iso-surface of error above 0.1% between the temperature reported by the two models after 45 times steps.

Figure 15 yields a great deal of information about the source of the underlying error. One region of error can be seen (marked in the oval region) in the bottom of the three dimensional space, which corresponds to the lower levels of the atmosphere. Similarly, another marked region can be seen in the upper levels of the atmosphere. Because of the underlying numerical scheme used to compute these quantities, it is impossible for both errors to be caused by the same fault, and thus they must be generated by two independent causes. After some further investigation it was discovered that the top error surface was due to differences in the radiation code of the two models, and the bottom region related to differences in the planetary boundary layer physics.

Armed with the approximate location of the errors, it was possible to construct a number of assertions which refined the region of error as in the previous studies. This investigation showed that the two models were inconsistent in two separate pieces of code. In one case, a source modification that was applied to MM5 has not be applied to MPMM. In the other case, MPMM was computing one of the quantities using a different numeric scheme, one

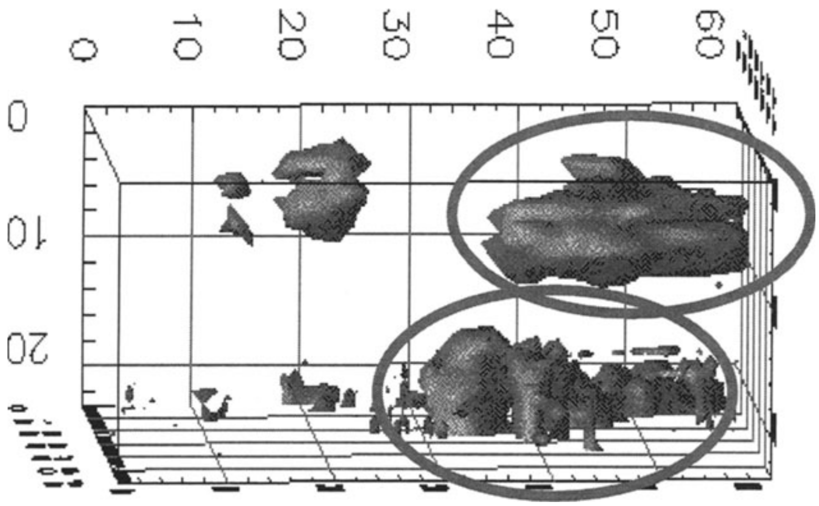


Figure 15. Multiple errors in global circulation model.

which was better suited for parallel execution. Whilst the scheme used in MPMM was numerically correct it still generated slightly different results to the one used in MM5.

This case study highlights a number of interesting aspects of GUARD. First, visualising the error surface can yield very important information about the nature of the underlying error and its locations. Second, when multiple independent errors are present, the visualisation is necessary to determine whether the assertions have passed or failed, rather than a simple error metric. Third, the two programs can have quite different internal structure, and it is still possible to compare key data structures. The case study is discussed in more detail in (Abramson et al., 1995).

6. Conclusions

In this paper we have described a new tool which supports the debugging and testing of programs developed with evolutionary software engineering techniques. The tool makes use of previous versions of a program because it allows comparison of data structures between new and old versions. Since it operates in a distributed heterogenous computing environment it is ideal for use in program porting because the original implementation can act as a reference site. Through a number of case studies, we have illustrated the power of the system.

GUARD is also useful for automatically testing new versions of programs against existing ones. The declarative assertion makes it possible to specify a number of assertions about key data structures before any changes are made to the code, and the program can be executed under the control of the debugger to verify that these assertions are met. More details about the implementation of GUARD can be found elsewhere (Sosic and Abramson; Abramson, Sosic, and Watson).

GUARD has been ported successfully to a range of sequential platforms, namely SUN, Next, DEC Alpha, IBM RS6000 and Silicon Graphics machines. This is a substantial achievement because each of these machines use different computer architectures and support debugger software in different ways. Further, because GUARD can compare data between systems it is necessary to convert exact data formats between systems automatically. In one of our case studes we illustrated the utility of this mode of opertion for finding subtle differences in system software.

Current research involves expanding GUARD to support the testing of parallel programs, supporting more data types and increasing the range of data structure visualisations which are possible. We are currently building a parallel version which can control the multiple processes of a parallel application. This has required us to redesign the logic used in the debugger for evaluating assertions, and this is discussed in another paper (Abramson et al.). The current version of GUARD interfaces with external visualisation systems by writing the data to files, which must then be processed by a separate data extraction program before the data can be visualised. It should be possible to define a higher level interchange format for this file which makes the process of setting up new visualisations easier for the user, and this is worthy of further consideration.

All of the functionality which has been described as part of GUARD could be intergrated into existing debugger software. The key requirement is that the debugger implements a client-server architecture so that it can control programs on more than one computer system.

This approach would allow the technology inherent in GUARD to be used as part of a much wider CASE environment, providing software developers with a powerful debugging and testing system. The authors are currently actively pursuing this approach with a number of computing vendors.

Acknowledgments

This work has been sponsored by the Australian Research Council. The authors wish to acknowledge the work of Lisa Bell, who has performed most of the programming necessary to implement GUARD. Thanks go to Simon Wail from IBM and Professor Geoff Dromey for proof reading a draft of this paper, and to Ian Foster and John Michalakes from Argonne National Laboratories for their work in applying GUARD to some real world scientific codes. Thanks also go to the CASE-95 program committee and the anonymous referees for their helpful comments.

Notes

1. Patent Pending.
2. The current limit is set to that the process always blocks after its breakpoint has been encountered. This appears to be satisfactory for many cases.

References

Abramson, D., Foster, I., Michalakes, J., and Sosic, R. 1995. Relative debugging and its application to the development of large numerical models. *IEEE Supercomputing*, San Diego.

Abramson, D.A., Dix, M., and Whiting, P. 1991. A study of the shallow water equations on various parallel architectures. *14th Australian Computer Science Conference*, 6:1–12, Sydney.

Abramson, D.A. and Sosic, R. 1995. A debugging tool for software evolution. *CASE-95, 7th International Workshop on Computer-Aided Software Engineering*, Toronto, Ontario, Canada, pp 206–214. Also appeared in *Proceedings of 2nd Working Conference on Reverse Engineering*, Toronto, Ontario, Canada.

Abramson, D.A., Sosic, R., and Watson, G. Implementation techniques for a parallel relative debugger, to appear, *Internatinal Conference on Parallel Architectures and Compilation Techniques—PACT'96*, October 20–23, 1996, Boston, Massachusetts, USA.

Adams, E. and Muchnick, S. 1986. Dbxtool: A window-based symbolic debugger for Sun workstations. *Software-Practice & Experience*, 16(7):653–669.

Anthes, R. 1986. 1986 summary of workshop on the NCAR community climate / Forecast models. *Bull. Amer. Meteor. Soc.*, 67:194–198.

Cheng, D. and Hood, R. 1994. A portable debugger for parallel and distributed programs. *Proceedings of Supercomputing 94*, Washington, DC, pp. 723–732.

Foster, I. and Michalakes, J. 1993. MPMM: A massively parallel mesoscale model. In *Parallel Supercomputing in Atmospheric Science*, Geered-R Hoffmann and Tuomo Kauranne (Eds.), pp. 354–363. World Scientific, River Edge, NJ 07661.

Galbreath, N., Gropp, W., and Levine, D. 1993. Applications-driven parallel I/O. *Proceedings Supercomputing-93*, Portland, Oregon, pp. 462–471, IEEE.

Grell, G., Dudhia, J., and Stauffer, D. 1994. A description of the fifth-generation penn state/NCAR mesoscale model (MM5). Technical Report NCAR/TN-398+STR, National Center for Atmospheric Research, Boulder, Colorado.

Linton, M. 1990. The evolution of Dbx. *Proceedings of the Summer 1990*, USENIX Conference, pp. 211–220.

McRae, G.J., Russell, A.G., and Harley, R.A. 1992. CIT photochemical airshed model—Users manual. Carnegie Mellon University, Pittsburgh, PA and California Institute of Technology, Pasadena, CA.

Michalakes, J., Canfield, T., Nanjundiah, R., Hammond, S., and Grell, G. Parallel implementation, validation, and performance of MM5. In *Parallel Supercomputing in Atmospheric Science*, World Scientific, River Edge, NJ 07661.

Moher, T. 1988. PROVIDE: A process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6):849–857.

Olsson, R., Crawford, R., and Ho, W. 1991. A dataflow approach to event-based debugging. *Software-Practice and Experience*, 21(2):209–229.

Ramsey, N. and Hanson, D. 1992. A retargetable debugger. *Proceedings of SIGPLAN'92 Conference on Programming Language Design and Implementation*, pp. 22–31, ACM.

Satterthwaite, E. 1972. Debugging tools for high level languages. *Software-Practice and Experience*, 2(3):197–217.

Sosic, R. and Abramson, D.A. GUARD: A Relative Debugger, to appear, *Software-Practice Experience*.

Sosic, R. 1995. *Design and Implementation of Dynascope, a Directing Platform for Compiled Programs*. Computing Systems, Spring, 8(2):107–134.

Sosic, R.A. 1995. *Procedural Interface for Program Directing*, Software-Practice and Experience, 25(7):767–787.

Stallman, R. M. and Pesch, R. H. 1994. *Debugging with GDB*. Free Software Foundation, Boston.

Stevens, W.R. 1990. *Unix Networking Programming*. Englewood Cliffs, NJ: Prentice-Hall.

Sun Microsystems, 1990. *Debugging Tools Manual*. Sun Release 4.1.

Automated Software Engineering 3, 391–393 (1996)
© 1996 Kluwer Academic Publishers. Manufactured in The Netherlands.

Desert Island Column

KEVIN RYAN kevin.ryan@ul.ie
College of Informatics & Electronics, University of Limerick, Ireland

The requirement remained deceptively simple. "Imagine you are marooned on a desert island, with only a handful of books and papers related to automated software engineering at your disposal. Which books or papers should those be? They may be seminal, thought-provoking or simply a pleasure to read".

Of course my first reaction was "why would anyone in their right mind want any software engineering books or papers at all under those circumstances?" You would bring a few survival manuals and possibly some classical literature but software books, research papers, theses—definitely not. But that is to miss the spirit of the challenge. Assuming the lower levels of Maslow's hierarchy of needs have all been catered for, the spotlight then falls on the word "related". What is or isn't related to "automated software engineering"—or indeed to anything at all? Why would you expect to do after your enforced sabbatical? So—paraphrasing furiously—I restate the requirement as: "Supposing you were to be marooned on a comfortable island for a year or more, in the full knowledge that at the end of that period you would return to work on an automated software engineering project, what would you want to read in preparation?".

My choice of three books and two papers is a mixture of the predictable and the eclectic.

Douglas Hofstadter's Gvdel, Escher, Bach (Hofstadter, 1979);
Henry Petroski's To Engineer is Human (Petroski, 1985);
Edward Tufte's The Visual Display of Quantitative Information (Tufte, 1983);
Cavalli-Sforza's Genes, Peoples and Languages (Cavalli-Sforza, 1991);
Joe Wiezenbaum's The Myths of Artificial Intelligence (Weizenbaum, 1983).

My reasons for choosing them may need some explaining.

The arguments for a classical education are frequently misunderstood. Some people imagine that we should reflect on past civilisations because they have all the answers to life's deepest questions or that, just as English has sprung from Latin, so our civilisation owes its existence to the Greeks and the Romans. But that is to miss the point. The ancients did not, by any means, have all the right answers. What they had was all the right questions. Every ancient civilisation faced the same human predicament. Questions of right and wrong, purpose and intent, truth and beauty are the stuff of every philosophy. In reflecting on how I might approach a new research undertaking I would like to draw on those questions and establish a framework of values that give perspective to my work and help keep me on the right track.

The first value is the aesthetic. How often have you heard dismissive comments such as "It's just equivalent to First Order Logic" or "The Human Computer Interface can be added later" which betray a lack of concern for the usability, the human-centredness, the sheer beauty of a research artifact? Presentation is not everything but, when it comes to providing useful tools, it is very important, and quick and dirty often hides muddled and misguided. That is why I would want Tufte along. This is a book beyond price. A comprehensive survey that will convince even the least aesthetic among us of the genius that is involved in describing complexity with clarity, accuracy and beauty. Minard's shocking graphic of Napoleon's retreat from Moscow is a classic and, according to Tufte, it may be "the best statistical graphic ever drawn". He also nominates, on page 118, a 'worst graphic' but, overall, the book is crammed with minor masterpieces and gems of simple wisdom. Absorb this book and thereafter timetables, graphs and charts will never look the same to you again, and you will demand that your own work exhibits clarity, economy and the indefinable quality called style.

Maybe we have to write too many research proposals or personal biographies but humility can be in short supply among software researchers. Petroski's book—subtitled "The Role of Failure in Successful Design"—is a calm but candid assessment of some classic disasters in civil engineering design. He argues that the engineer must anticipate failure, do everything possible to avert it and then, when failure inevitably occurs, have the courage and the humility to learn as much as possible from the post mortem. As for computers in design, he considers them "both a blessing and a curse". A blessing because they take away the tedium of calculation; a curse because they can so distance people from physical reality that they lose the ability to sense when an answer is "unreasonable". More pointed by far is Weizenbaum's withering critique of pride and delusion in artificial intelligence research. He parses and dissects, to great effect, quotes old and new. Prospective 'knowledge engineers' would do well to remember Simon and Newell's 1958 prediction that "within the visible future—the range of problems [computers] can handle will be coextensive with the range to which the human mind has been applied". Of course some people are long-sighted but surely we've reached the visible future by now. Feigenbaum and McCorduck are quoted as asserting that "no plausible claim to intellectuality can possibly be made in the near future without an intimate dependence upon this new instrument" [the computer] and that "the burden of producing the future knowledge of the world will be transferred from human heads to machine artifacts", while Moto-oka of the Fifth Generation project believed that "through the intellectualization of these advanced computers, totally new applied fields will be developed, social productivity will be developed, and distortions in values will be eliminated". Weizenbaum found the last idea particularly reprehensible. But it is not so much that the predictions of 1958, 1983 or 1996 were and are wrong, since that is the nature of predictions. It is that they betray the arrogance of an isolated elite. Who are we to assume that what we do supercedes all previous human endeavor? To imagine that the big questions of the ancient philosophers have somehow been solved by our superfast calculations? Some readings on humility are definitely in order.

Integration is the last of my basic values for researchers. Hofstadter's book was a landmark fusion of scientific method, whimsical speculation and subtle humour. Weaving his "eternal golden braid" out of the recurring patterns of music, biology and computing, he showed the limitations of logic and the layered nature of all meaning. Most computer scientists have bought this book. Many may have even read it, but everyone can continue to enjoy dipping into it—even without the benefits of a tropical paradise. And it is to the tropics that my last, and most fascinating, author traces our origins. Cavalli-Sforza's short paper in the November issue of Scientific American literally took my breath away. It tells how the genetic mapping of the human family tree can be shown to correspond closely with the postulated tree of human languages. He traces the movement of our earliest ancestors from eastern Africa to all corners of the globe and, in the process, solves many linguistic conundrums and raises a few more. Almost audibly the chunks of disparate knowledge click into place and the whole picture, the story of humanity's spread, is laid out with compelling logic and no little beauty. The theory is not uncontested. Perhaps Cavalli-Sforza will even be proven wrong. But I will take his paper with me as a model of the scientific goal of unearthing the simple patterns which underly observed complexity.

So there you have it. Three virtues to cultivate prior to returning to the grant-hunting, deliverable-driven fray. Beauty, humility and integration but, if I have to choose, the greatest of these is beauty.

References

Cavalli-Sforza and Luigi Luca. 1991. Genes, peoples and languages, *Scientific American*. 265(5):72–78.
Hofstadter, D. 1979. *Gvdel, Escher and Bach*. Basic Books.
Petroski, H. 1985. *To Engineer is Human*. St Martin's Press.
Tufte, E. 1983. *The Visual Display of Quantitative Information*. Graphics Press.
Wiezenbaum, J. 1983. *The Myths of Artificial Intelligence*. The New York Review, Reprinted in T. Forrester, (Ed.), 1985, The Information Technology Revolution, Blackwell.

Table of Contents: Volume 3 (1996)