

BUSINESS COMPONENT-BASED SOFTWARE ENGINEERING

*Edited by
Franck Barbier*



Springer Science+Business Media, LLC

**BUSINESS
COMPONENT-BASED
SOFTWARE ENGINEERING**

**THE KLUWER INTERNATIONAL SERIES
IN ENGINEERING AND COMPUTER SCIENCE**

BUSINESS COMPONENT-BASED SOFTWARE ENGINEERING

edited by

Franck Barbier
Universite de Pau, LIUPPA
France



SPRINGER SCIENCE+BUSINESS MEDIA, LLC

Library of Congress Cataloging-in-Publication Data

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

Barbier, Franck

Business Component-Based Software Engineering

ISBN 978-1-4613-5429-1 ISBN 978-1-4615-1175-5 (eBook)

DOI 10.1007/978-1-4615-1175-5

Copyright © 2003 by Springer Science+Business Media New York
Originally published by Kluwer Academic Publishers in 2003
Softcover reprint of the hardcover 1st edition 2003

All rights reserved. No part of this work may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, microfilming, recording, or otherwise, without the written permission from the Publisher, with the exception of any material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work.

Printed on acid-free paper.

TABLE OF CONTENTS

Preface	vii
1. Business Components	
Franck Barbier and Colin Atkinson.....	1
2. Model-Driven, Component-Based Development	
Colin Atkinson and Hans-Gerd Gross.....	27
3. SCARLET: Light-Weight Component Selection in BANKSEC	
Neil Maiden and Hyoseob Kim.....	49
4. Built-In Contract Testing for Component-Based Development	
Hans-Gerd Gross, Colin Atkinson, Franck Barbier, Nicolas Belloir and Jean-Michel Bruel.....	65
5. Interfaces and Techniques for Runtime Testing of Component-Based Systems	
Jonathan Vincent, Graham King, Peter Lay and John Kinghorn.....	83
6. The NEPTUNE Technology to Verify and to Document Software Components	
Juan Carlos Cruellas, Jean-Paul Bodeveix, Thierry Millan and Agusti Canals.....	101
7. The OOSPICE Assessment Component: Customizing Software Process Assessment to CBD	
Friedrich Stallinger, Brian Henderson-Sellers and John Torgersson.....	119
8. The OOSPICE Methodology Component: Creating a CBD Process Standard	
Brian Henderson-Sellers, Friedrich Stallinger and Bruno Lefever.....	135
9. QCCS: Quality Controlled Component-Based Software Development	
Torben Weis, Noël Plouzeau, Gabriel Amorós, Petr Donth, Kurt Geihs, Jean-Marc Jézéquel and Anne-Marie Sassen.....	151

10. Components for Embedded Devices: The PECOS Approach	
Thomas Genssler, Alexander Christoph, Michael Winter and Benedikt Schulz.....	167
11. Model-Based Risk Assessment in a Component-Based Software Engineering Process: The CORAS Approach to Identify Security Risks	
Ketil Stølen, Folker den Braber, Theo Dimitrakos, Rune Friedriksen, Bjorn Axel Gran, Siv-Hilde Houmb, Yannis C. Stamatiou and Jan Øyvind Agedal.....	189
12. A Vocabulary of Building Elements for Real-Time Systems Architectures	
Jose Luis Fernández-Sánchez.....	209
13. COTS Component-Based System Development: Processes and Problems	
Gerald Kotonya, Walter Onyino, John Hutchinson, Peter Sawyer and Joan Canal.....	227
14. Component-Based Software Measurement	
Yingxu Wang.....	247
Biographies	263

PREFACE

Component-Based Software Engineering (CBSE) is increasingly becoming a significant focus of research in academic areas of computer science as well as in the software industry. Despite the progresses relating to object-oriented programming, design and modelling during the '90s, reuse and reusability of software entities are nowadays slowed down by new factors. Thus, the size of components, their deployment capability, their compatibility and interoperability features with respect to their incorporation into heterogeneous distributed environments, their faults tolerance and, above all, their ability to fulfil performance conditions are all critical and crucial expectations of software engineers.

The idea of a component in epistemology is old and is linked to Descartes' method in the 17th century: "diviser chacune des difficultés que j'examinerais en autant de parcelles qu'il se pourrait et qu'il serait requis pour mieux les résoudre". In English, this means that the parts ("parcelles") of a problem are to be divided into smaller ones until a way is found to easily and efficiently "solve" them. Unfortunately, solving all parts one by one (recursively) is not equivalent to solving their sum resulting from their integration. Applying this principle in software engineering supposes that parts are thus separated but equipped with specific properties so that their future assembly is really optimised. Object-orientation has brought new ideas on this, especially with the notion of encapsulation: parts are endowed with some protection in order to avoid undesired side-effects relating to their mutual interaction. Nevertheless, object-oriented components like classes or, at a higher level of granularity, software aggregates and patterns in OO, cannot be considered as the overall problems of CBSE.

The evolution of the software market has made emergent many kinds of software parts which do not always conform to the theory of object-orientation: for instance, binary components such Windows' DLLs. On the other hand, component paradigms like CCM (CORBA Component Model), JavaBeans and Enterprise JavaBeans have enriched the original spirit of object-orientation. Besides, some normative approaches have advocated canonical and implementation-free component models: e.g. CORBA components essentially written in IDL and implemented by means of various programming languages.

Despite the maturity of the component market and some available results coming from academic research, CBSE remains, somehow, a backward discipline, since no modern techniques for component composability have been recognised as satisfactory. Indeed, even if components are coherent software units that are in essence encapsulated, their combination is not

straightforward: this has led to design practices that assume components are compositional in all behaviours. This is obviously not true.

This book aims to complement other reputable books on CBSE, by stressing how components are built for large-scale applications, within dedicated development processes, for easy and direct combination and to leverage high-confidence software systems. The book “Business Component-Based Software Engineering” emphasises these four facets and offers a complete overview of some recent progress. Projects explained here are intended to prompt graduate students, academics, software engineers, project architects, managers and developers to adopt and to apply new component development methods gained from and validated by the authors of this book *who are academics and professionals that shared their experience by working on the same projects.*

In this context, the book recaptures and analyses the history, the current situation and the challenges of CBSE based on the state of the art assessment presented in chapter 1: What really are business components? Chapter 2 focuses on the KoBRA method for component development that customises and adapts the concept of “model-driven design” of components. This extends the OMG’s idea of Model-Driven Architecture leading to pay attention and to study architectures in which components are deployed.

Chapters 3, 6, 7, 8 and 13 give some new insights concerning tailored software development processes and tools for component development. Chapter 3 especially illustrates a proposition in the banking domain while the other chapters treat the documentation issues, the human aspects (how teams are organised in component development) as well as the relation between component development and analysis/design methods, as well as requirements engineering.

Another strong requirement for components is reliability. Although, they may offer facilities to be connected with each other, components may fail as a result of some misuse or by breaking some contract. Chapters 4, 5, 9, 11, and 14 supply innovative techniques in order to incorporate additional material in components: this corresponds to an increasing quality, and a way to test, sometimes certify and surely improve, component composability. This also enhances their flexibility to be deployed in unknown (at the development time) target environments. A special interest for security-critical systems and real-time systems is developed in chapters 10, 11, 12.

Pau, June 3, 2002

Franck Barbier

Chapter 1

Business Components

Franck Barbier and Colin Atkinson

LIUPPA, University of Pau, France; IESE, Germany

Abstract: Business components are software assets modeled within requirements engineering and domain engineering activities. They embody business logic, rules and constraints whose consistent grouping generates reusable, compositional, highly abstract modules that are specific to a domain. Business components therefore migrate through the software development cycle and are connected, via a continuum, with technical components that encapsulate software facilities such as primary data structures, generic algorithms, common services (e.g. dating, timing, graphics user interface, network communication, data storage). The increasing demands on business components driven by the goal of a component market require a new set of engineering practices. This chapter looks at the types of engineering methods that are suitable for designing business components. To this end, business components are carefully characterized, distinguished from the other kinds of components, and finally exemplified.

Key words: Software components, business engineering, component modeling, component standards

1. INTRODUCTION

The idea of software components cannot really be regarded as new because prominent books, such as that of Szyperki [1], have already highlighted most of the technical features, issues, challenges and trends of Component-Based Development (CBD). More generally, Component-Based Software Engineering (CBSE) is recognized as a fully reviewed, and thus mature, software technology [2] [3].

The number of component standards, environments/platforms, products and development methods is currently in a stage of rapid growth, demonstrating constant expectations concerning the component paradigm. In particular, component-dedicated development methods, as for instance Catalysis [4] or Kobra [5], have recently emerged to address the need for a new set of engineering practices for CBD. This raises the specific and major technical problem of dealing with *external sources* within the development process. Due to the existence of Commercial Off The Shelf (COTS) components, it is difficult to express the use and the integration of such imported, predefined, even inflexible, software elements in specifications, designs and implementations. The other major difficulty is linked to deployment which raises distribution concerns which cannot be addressed in a straightforward manner. Because components differ from objects, particularly in their granularity, packaging objects into modules that are deployable calls for some originality in both modeling and programming languages.

The need for a general-purpose and consensual formalism for components is obvious. These may be COTS components or in-house components. These may also be components that are close to user requirements concerns or to more or less detailed implementations. Components can, in fact, vary on an abstraction scale. In order to use and often customize well-known modeling languages, such as the Unified Modeling Language [6], most authors have proposed UML-centered development approaches [4] [7] [8] pp. 243-262 and [5] for CBSE in all kinds of development contexts. In this chapter, we deliberately stress the notion of business component, because assembling business components from home-made or COTS components such as, for instance, those based on the .NET platform is inherently challenging. We require a continuum in space and time to manage each varied dimension of components. The business concerns, strongly linked to the applications domains, are the driving concerns since they appear early and recurrently among applications.

Modularizing business logic and, more generally, vertical application domain assets for reuse is one thing, but respecting standards of design and making business components operational is another. Moreover, smoothly transforming business components into code and finally composing component architectures by means of odd, multiple-source, multiple-shape components is the hardest problem of all. All of these tasks (i.e. building Enterprise Java Beans, CORBA components, .NET modules or whatever on a day-to-day basis) are not yet widespread practices. In short, the observation that the adoption of CBSE is slow, is supported by real-world experience and economic surveys [2]. Easily comprehensible business components that are familiar mental concepts to non-computer experts may thus help speed up the adoption of CBD in industry.

This chapter aims to concretely discuss the business component concept, showing and verifying that business components are, above all, *true* components (i.e. reusable entities and are by definition compositional in all situations). In addition, they are scalable, deployable in complex networking infrastructures, and necessarily lead to the creation of predictable and environment-flexible assemblies.

Unfortunately, however, there is an embarrassing contradiction in these assertions. From a strict technical point of view, the separation of a component's interface from its implementation certainly favors reusability. However, considered as a syntactical emergent property, the overall interface of a component does not guarantee the component's continued compliance with and adaptation to its environment. In other words, hiding implementation precludes an understanding of the component's fine-grained behavior. From the system viewpoint, components have to be resilient to faults, for example, and thus have to be designed for various environmental contexts. From the business and user's side, however, components are subject to changes in requirements that are difficult to foresee. Implementing domain logic in software artifacts invariably makes it difficult to isolate "modules" that need to be integrated in systems for which they were not originally constructed. For instance, merging financial components that cope with legal issues of different countries into one single application is a non-trivial task. Its complexity depends on whether, for example, such components are endowed with technical security management mechanisms, which could be incompatible with the underlying networking security policy.

Thus we are faced with the great contradiction of trying to find a conceptual model of software components that possesses all the desired software engineering qualities (e.g. robustness, reusability, adaptability, configurability), but can, at the same time, support complex domain "intelligence". Because business components must be usable and scalable, they are both units of decomposition (particularly at the time when they are discovered) and composition. Designing business components is above all a technique of rationale segmentation that occurs either early or concurrently with regard to the software development cycle.

In this chapter, after offering a characterization of business components, including adoption inhibitors, we list and analyze the interesting properties that business components have to possess. We talk about their executability, their qualification and adaptation, their seamless development and connection with technical components and, of course, their reusability. At the end of the chapter, we consider the example of the Currency business component from the Object Management Group (OMG). Focusing on the use of UML during the course of this chapter, we take the opportunity to suggest some amendments for CBD.

2. THE CONCEPT OF BUSINESS COMPONENTS: A CHARACTERIZATION

The expressions “business object” or “business component” are admittedly appealing but previous attempts to characterize these concepts [9] highlight the lack of a common understanding. Business components, when regarded as ordinary components, are preferably methodology-neutral or technology-free. For example, within a vertical application domain such as healthcare we have business components (formal descriptions, behavioral models, runtime forms, etc.) insofar as we can identify abstractions that are reusable from one application to another. Unfortunately, however, not all applications can easily share the same modules. For example, applications can belong to different corporate entities or distinct departments (e.g. two hospitals or two divisions such as dental surgery and plastic surgery). Commonalties actually may exist but they affect the area of healthcare according to very different ways.

The problem is that boundaries between domains are often unclear. Consequently, it is difficult to establish the scope of a business component. In other words, to what extent, does it gain the status of a modular software artifact in a domain or sub-domain? Domain analysis with the objective of discovering reusable elements is discussed in [10]. In this chapter, we restrict the characterization of business components to well-known domains described in standards or books. Standards favor consensus and convergence by enabling the agreed delimitation of domains and thus the factorization/formalization of significant components. As an illustration, the General Ledger business component produced by the OMG [11] is based on international accounting standards. The acknowledgement of such a standard for European and American companies will probably allow the interoperability of most of their respective accounting programs.

Standardization of business components has mostly occurred in relation to the efforts of the OMG to develop CORBA and its associated CORBA Component Model (CCM). Domain component specifications are nowadays available at the OMG in the areas of healthcare, simulation, utility management (covering water and energy management systems), transportation, life sciences research, manufacturing, finance, telecommunications and electronic commerce, with sub-domains such as air traffic control for example in the Transportation field (see: http://www.omg.org/technology/documents/domain_spec_catalog.htm).

Other initiatives, not linked to a software standardization group like the OMG but based on a collection of leading companies in a given market, also exist. In the oil and gas industry (see: <http://www.openspirit.com/>) for instance, a CORBA framework offers dedicated business components that are usable through a unique component infrastructure.

Such convergence efforts do not have to hide the fact that business components also exist outside recognized standards. The list of domains of expertise under the authority of the OMG to define mature and broad vertical application CORBA components is insufficient. Other interesting growing fields have not yet been sufficiently investigated and stumble against the problem of capturing key components within sub-domains. Educational software components are one of the next promising market niches. In particular, [12] categorize educational components in terms of their strong cognitive characteristics compared to their weak computational power. Educational business components are mental imitations of human learning tasks as well as of synchronized teaching/learning activities.

Business components are therefore today very real!

2.1 Definition

A business component models and implements business logic, rules and constraints that are typical, recurrent and comprehensive notions characterizing a domain or business area. Within software engineering, business components are key abstractions that are captured during the domain engineering activity. They are software artifacts in the sense that they are *not* part of reality but embody and represent recurrent invariants relevant to requirements, particularly at the earliest phases of development. As such, they are transformed into concrete code for the software development process in order to acquire operational features, especially for the purpose of being connected with more technical software components that embody the underlying facilities and services within a software system.

2.1.1 A business component *is* a component

Szyperski noticed that “Components are for composition” [1] p. 3 which invariably implies that talking about *reusable* components is a pleonasm. Any implicit use of the word “component” instead of “part”, “chunk”, “piece” or whatever, within the worldwide software engineering community, supposes and imposes reusability [13]. However, incorporating components into software architectures and frameworks shows that the reuse power of components is handled with difficulty. As observed in [14], “However, this has led to design practices that assume components are “compositional” in all other behaviors: specifically, that they can be integrated and still operate the same way in any environment. For many environments, this assumption does not hold.”

All of these statements necessarily apply to business components. As a “traditional” component, a business component is a self-contained software entity with well-defined interaction points facilitating the accessing and

execution of a coherent package of functionality. “Coherent” here means that business logic, rules and constraints are separated into highly cohesive and loosely coupled elements. For instance, a coarse-grained Enterprise Resource Planning (ERP) business component has to be isolated and to encapsulate discriminating functionality with respect to a Customer Relationship Management (CRM) component. A high-level invoicing function within an application may thus invoke both components, but based on communication and data exchange between the two rather than on interrelating sub-functions. In some circumstances, each must be deployable by third parties, ignoring the other.

In general, the component concept can have both a runtime and a development time incarnation [1], [5]. Component approaches cope with these aspects in different ways. In the rest of this chapter, we use Szyperski’s explicit distinction between component (i.e. at the type level) and component instance. This is depicted in Figure 1 in UML where the concept of component, and its specialization business component is a *kind of* “Classifier”.

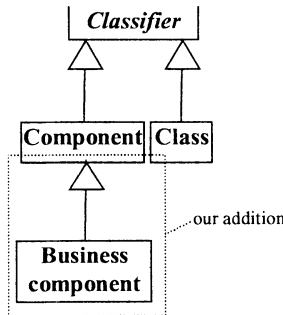


Figure 1. Component and business component concepts within the UML metamodel

2.1.2 Foundations of business components

Business components involve the juxtaposition of two software technologies: that of Domain Engineering / Domain Analysis and that of Component-Based Software Development / COTS Integration. These technologies are carefully reviewed and synopsized at <http://www.sei.cmu.edu/str/descriptions/deda.html> and <http://www.sei.cmu.edu/str/descriptions/cbsd.html>, respectively.

Domain engineering refers to a single self-contained and self-consistent software technology. It covers the process of defining the domain, analyzing the domain, developing the domain architecture and building the components for a class of multiple applications. Domain component engineering focuses on designing domain assets for reuse [10] on a continuous development scale,

from a conceptual level (modeling) to an operational one (implementation, deployment). Business components are artificial entities that embody, and therefore imitate, any prominent distinctive knowledge and know-how within a particular field (or a family of fields) of interest. Nevertheless, it is essential *not to restrict* business components to models, as it is sometimes the case in business engineering [15]. Business component specifications are seamlessly refined into code and binary blocks for execution at runtime and thus appear in multiple, inter-related forms. This happens during design activities involving the combination of business components and technical components. The latter's services are used for the implementation of the business components' services.

The foundations of business components do not originate solely from object orientation but the similarities are numerous. This section aims to add some definitional elements on, in particular, the difference between objects and components in their business guise. However, the strict difference between components and objects remains somehow “shaky” in the literature. Neither the UML [6] pp. 3-170-3-178 nor Szyperski really clarify a formal distinction.

Object-oriented programming (or modeling) provides one foundation for component programming (or modeling) – but not the only one. A business component is a server of a limited set of object types. Within a business component specification, several object types are defined that are either internal (used for internal computation) or external (intended for the use of clients). In the latter case, they participate in the signatures of the operations making up an interface of a business component. The entry point of a business component is often implemented through an object type or class. Well-identified “provided” and “required” interfaces facilitate the neat isolation of a business component's “micro-architecture” requiring at best the definition of a small number of object types.

Instances of object types at runtime augment functionality while instances of business components enhance quality of service. A business component instance is deployed once and is *a priori* unique in a component framework. In the domain of manufacturing, for instance, a Data Acquisition (from industrial control processes) business component instance is not duplicated: all the desired operations are present in the application using it. In contrast, objects of the same type that have their own state and represent physical entities on the supervised processes are replicated according to functional and behavioral constraints at runtime.

2.2 Characterizations by other authors

In [8] p. 83, Williams introduces the notion of “domain components” as “Domain components are what most developers think of when they talk about

business components. They are reusable or not. (...) For example, the domain components in an insurance application could include Bill, Policy, and Claim components.” He especially distinguishes “Domain components” from “Service components” and “GUI components” by means of a cost/complexity scale. The complexity and development cost of domain components are higher.

In the same book [8] p. 286, Carey and Carlson describe the concept of business component as follows: “A business component is a software component that provides functions in a business domain. (...) for example, an order management application is part of the Enterprise Resource Planning (ERP) business domain.” They insist on the difference between fine-grained and coarse-grained components and mainly discuss the development of coarse-grained components based on other coarse-grained components and, according to their rationale, on fine-grained components that are similar to classes in programming languages. Although UML is used as the component modeling language, Component Diagrams are noted used; instead Class Diagrams are used as a substitute.

Older contributions, such as that of Fowler [16] do not exactly use the term “component” but, apart from this terminology subtlety, the idea of business component is easy to see: “Analysis patterns are groups of concepts that represent a common construction in business modeling.” [16] p. 8. Since the key power, of business components is reusability and analysis patterns possess this power they are candidates, at implementation time, to become components. Fowler provides many examples of reusable business entities and, in addition, gives some implementation hints for three-tier architectures based on the concept of “support patterns”.

Finally, Kilov in [15] pp. 131-189 evokes “business patterns” in discussing business-specific and business-generic patterns. He also called these: “common business components”.

Fowler and Kilov implicitly view business components as conceptual entities compared to software (i.e. runtime) entities. More practical characterizations such as those of the OMG mentioned above are implementation-oriented since they are specified by means of the Interface Description Language (IDL) that directly maps to programming languages.

2.3 Non-business components

A recurring problem in science, particularly in software engineering, is categorization. Viewing things as discrete systems often preclude the characterization of relative and continuous concepts. Hence, a possible way to define what business components are, is to supply examples of what they are not. This may be helpful whether or not we consider the boundary between business components and *non*-business components to be clear and sharp

(Figure 2, left-hand side). Unfortunately, this is not justified in reality because the concept of business component is interpretation-prone (Figure 2, right-hand side). On the right-hand side of Figure 2, business components play roles with respect to a given domain: they therefore become software archetypes for that domain.

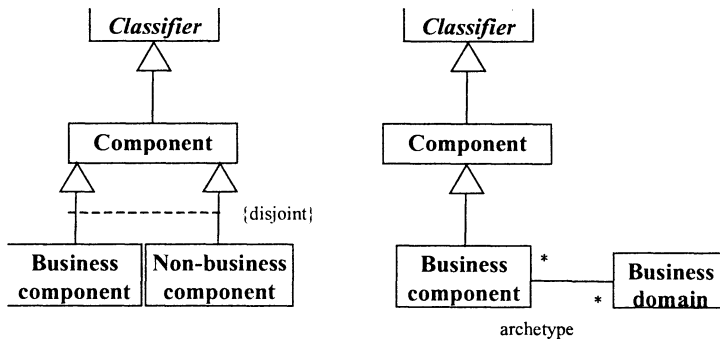


Figure 2. Two possible extensions of the UML metamodel enhancing the notion of business component

Business components that are accessed from several domains tend to lose their domain-centric features! The most interesting illustration of this is Fowler’s “Observations and Measurements” analysis pattern [16] pp. 35-55. He specifies dedicated business object types such as Phenomenon, Unit, Quantity, Measurement, Observation that collaborate together and operate, for instance, in such domains as healthcare (patient care recording and analysis). Such components may also apply for network management (quality of service recording and analysis, for example) and are probably useful in other domains (e.g. measurement engineering in mechanics).

The extreme case of a non-business component is a timing service including dating operations. This is known to be universal – and therefore may be regarded as a non-business component. As an illustration, a general-purpose Time Service component is offered in the Time Service Specification [17] that comprises a Timer Event Service sub-component. The Time Service component appears on the bottom of Figure 3 (left-hand side) with its *provided interface*. It also appears on the top of Figure 3 (left-hand side) with the Timer Event Service sub-component which is depicted *as part of the overall provided interface*.

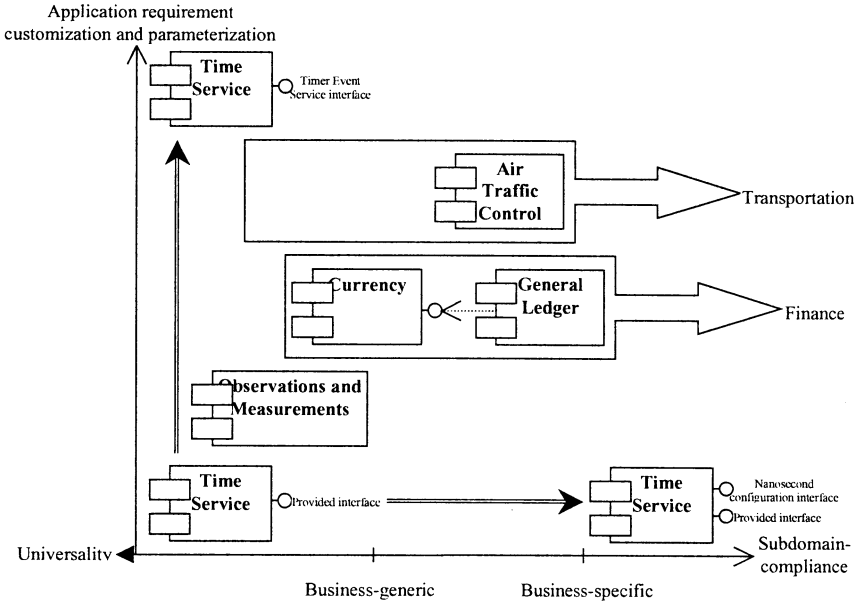


Figure 3. Continuity relating to the low/high business nature of components

In Figure 3, components may be qualified as “business” to the extent that they are positioned towards the right-hand side. Based on its cross-domain nature, the “Observations and Measurements” business component is, however, on the left-hand side of this figure. An interesting point is the application-based specialization of a component such as Time Service. In a real-time program for instance, only a subpart of its functionality is required. Hence, the customization of the Time Service component on the top of Figure 3 (left-hand side) occurs via a limited use of the provided interface, namely the Timer Event Service sub-component. At runtime, this may give rise to a reduction of the required resource.

A significant issue is also the specialization for a domain. The same Time Service component gains business features when it owns a configuration interface in order to support nanosecond precision (right-hand side, bottom of Figure 3). For example, nuclear process control applications compulsorily have such a requirement, which is often superfluous in other domains. The key idea linked to a configuration interface is that it controls the adaptation of a component to its environment. Nanosecond calculations cannot be supported *a priori* in signature declarations and, above all, in implementations as a consequence of high overheads. In contrast, a general-purpose Time Service component may exhibit such business characteristics, if required, through reconfiguration.

The boundary between business and non-business components is thus a fuzzy one. Even though domain and component taxonomies bring clarity and comprehensibility, this does not prevent business component from being tailored according to specific contexts.

2.4 Brakes and barriers

Although the notion of business component cannot be ignored, one has to be aware of factors that may obstruct the dissemination, and thus the use of, business components. In this context, people and country norms, habits, cultures, laws and legal issues in general, are all major forces on the nature of business software systems.

Obviously, internationalization of standards is a must for business component development. As an illustration, the OMG Air Traffic Control component [18] is based on such a premise. For security reasons, in the domain of air transportation there are many formal technical agreements that help the specification, design and implementation of such a normalized software subsystem. This is, however, not so easy in other areas (e.g. Education [12]), especially when business practices are ill-formalized. Standardization for vertical application components is said to be “*not so hard*” by Szyperski in [1] p. 38 because markets are narrow and involve few stakeholders. This, however, deserves further analysis.

Beyond the existence of software standards, the creation of business objects assume that domain standards exist. For instance, in the field of Customer Relationship Management (CRM), call/contact center applications may need voice infrastructure (e.g. telephony application programming interface) and dedicated software components. Standardization is trickier here. Local constraints may appear: spoken languages are odd and varied, recording conversations may be forbidden by law in given countries, etc. This raises the need for a high degree of parameterization in order to organize components with respect to such non-functional requirements.

Next, from a strict economical point of view, standards may preclude business competition by providing no way for software component vendors to increase the value of their own components and thus to improve their business advantage. In addition, standards are forever changing, if not dying! In [2], consumers complain about the instability of component standards, regarding them as the main inhibitor of CBD success. From the purchaser’s side, investment security is linked to widely agreed standards that evolve carefully and often slowly. However, this does not seem to be the current situation in the software world. From the seller’s side, rigidity, uniformity and strong dependency upon norms may be the risk for business components. Attractiveness not only resides in reusability and conformance to standards; providers also have to maximize the trustworthiness, context tuning,

verification/validation (testing) capabilities of their COTS business components. This imposes design-specific rules for such software entities that are discussed in the third section of this chapter.

Stable properties of business components are expected by consumers. These have to be connected with domain standards, if they exist, and with well-known reputable industry-consensual software tools (XML, Java, CORBA, .NET). Significant flexibility is also required for business components which may be split into the following concerns:

- Genericity: which implies that business components have to be highly parameterized;
- Configurability: which means a strong degree of adaptability;
- Predictability: which involves anticipating, measuring, assessing and mastering component assembly behaviors;
- Scalability: component engineering practices, which provide a uniform way of reasoning about components independently of their granularity.

3. COROLLARY FEATURES FOR BUSINESS COMPONENTS

The business properties of a component may be subject to varied interpretation due to its broad spectrum and differing business practices. As indicated at the beginning of this chapter, the idea of a COTS business component is problematic. In most traditional software development cultures, external parts in software are only service components, e.g. a relational or an object database engine, a Java package to access a middleware platform, a set of windowing classes or a security policy module. The day-to-day business logic, rules and constraints are typically encapsulated within in-house parts that connect with a common technical tool kit. There are, in fact, many risks involved in outsourcing predefined business components that have been constructed without considering how to match them to the contexts in which they are usable.

Component users are naturally reluctant to incorporate software parts into their products that fulfil domain demands but ignore application-specific concerns. Although this observation applies for non-business components as well, these allow the development of general purpose software components. Competing end user business applications may thus be too similar since they share a lot of business basis modules that have been supplied from the same source. Beyond this problem, application quality depends upon reused components. For business components, this can be embarrassing in the sense that the quality impression (e.g. usability) gained by end users is based on business component quality of service. Technical components that are buried

in applications have a lower impact on the end users' impression of business delivery.

3.1 Semantic contract versus syntactical interoperability

In this section, we discuss the paradox that makes business components general enough to be usable in a wide variety of contexts but, at the same time, domain-compliant and even business-specific. In particular, we compare and contrast the benefits of standards in proposing general-purpose design rules for business components.

Heiler notices in [19] that “Interoperability among components of large-scale distributed systems is the ability to exchange services and data with another.” Component interfaces are the basic foundation for this ability which can be viewed as ensuring the syntactical compatibility of components. However, syntactic compatibility does not ensure that the provided and required interfaces of two different components match in the semantic sense, nor that their full range of mutual collaboration possibilities are attained. For this reason, Heiler also outlines that: “*Semantic interoperability* ensures that these exchanges make sense – that the requester and the provider have a common understanding of “the meanings” of the requested services and data.” In other words, components must consistently interact with each other in the achievement of some well-defined *higher-level* computation. For this reason, many research projects on composability stress the “design by contract” idea of Meyer [20]. Composability means that potential client/server interaction are in semantics-compliance and, thus, do not just agree on the nature of the operations' signatures in the provided and required interfaces.

The contract mechanism, which is most fully developed in Eiffel, is not, however, easily generalizable for components. For example, typical representations of components using IDL only give syntactical entry points, namely the signatures of the interface's operations. This is obviously insufficient because business components such as the Air Traffic Control component [18] or the Currency component [21] are not easy to understand. They are documented in IDL with additional UML diagrams (sequence diagrams in particular for the Currency component). Unfortunately, even in UML, much progress has to be made to attain a complete component modeling language (see below). Basically, there is no rigorous way to express the semantic interoperability of components, except, maybe, through the use of the Object Constraint Language (OCL) in UML component specifications. This approach has not yet been sufficiently explored, however.

A crucial need for fully supporting the specification contracts is to be able to describe in what contexts executing the components will be safe and will lead to the expected behaviors and results. In general, this challenging

requirement covers the predictability and comprehensibility of component assembly behaviors.

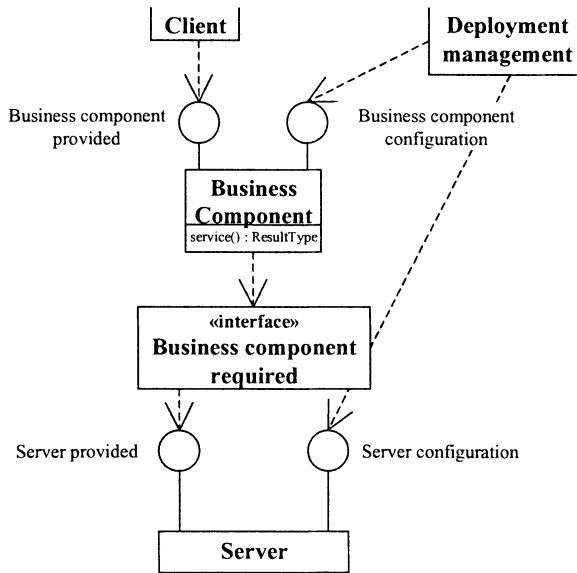


Figure 4. Business component canonical organization

As an illustration, Figure 4 proposes a minimal collaboration architecture for business components. A business component possesses a provided and a configuration interface (two top circles) and is dependent upon a required business component interface (as depicted in Figure 4) as a logical entity owning the “«interface»” UML stereotype. Contracts may therefore, for example, be statically formalized in OCL as follows:

Context Business component::business component provided::service() : ResultType

pre : - *a priori* conditions

post : - *a posteriori* conditions

The same kinds of constraints must also be developed in conjunction with behavioral specifications. This concerns the problem of the Currency component discussed above. Sequence diagrams document the use context of the Currency component, but no constraints to state how and why the next step in the scenario may apply, what the exact object instances exchanged are and what the logical states of the component are, etc.

3.2 Business components appropriation

Acquiring business components is not trivial. The trustworthiness of COTS business components can only be guaranteed under certain

circumstances. First, components that may be eligible must be appraised with regard to some given application requirements. This is the process of component qualification. Domain functionality offered by a candidate business component has to be validated, possibly even certified, by a recognized authority that requires the satisfaction of strict target criteria. For instance, business components supporting military or legal obligations must be carefully assessed. Next, component adaptation refers to the measurement and the realization of moderate tailoring changes to the acquired components. This covers the idea of flexibility evoked above. Quality of service (security, response delivery, robustness, etc.) has to be analyzed for components assembled into application architectures and component frameworks. Business components are built on top of more technical software components whose implementation may affect runtime resources. In this regard, performance corresponds to another important quality of service concern.

There are thus two angles to the process of appropriation. Components must first be qualified according to whether they are business-compliant. Through emergent well-formalized properties, the embedded business logic, rules and constraints in components must be fully understandable. Secondly, their implementation on sound and available underlying service components must be established. This involves the definition of how the values of business component parameters may ensure the satisfactory integration in an application architecture and/or component framework. For instance, a business component encapsulating basic manufacturing process control algorithms may have a parameter for setting up the memory management policy with a possible default value. A specific memory management sub-component may be a justified substitution for this default because the component's deployment node is an operating system with hard memory constraints. This may be sketched in C++ as follows:

```
template<class Technical_component_parameter, class Memory_manager =
Default_memory_manager<Technical_component_parameter> > class
Manufacturing_process_control_basic_algorithms {...};
```

In the code above, the possible late binding of an appropriate memory management service component demonstrates where the openness, and therefore the flexibility, exists. The default “Default_memory_manager” technical component may be replaced as a means of customizing the implementation of the wrapping business component named “Manufacturing_process_control_basic_algorithms”.

More generally, the business acceptance of components cannot just depend upon standards. Standards are necessary but not sufficient. Because business components are conceptual software artifacts, they must possess cognitive properties that enhance their understandability. In this respect, an orthogonal solution to standards is component executability. In this regard, we first quote a remark by Houston and Norris relating to UML in [8] p. 261: “For the area

of CBSE the most exciting initiative is the proposed introduction of action semantics. The objective of this proposal is to provide a mechanism to express the executable semantics of UML model elements in an implementation-independent fashion.” Next, recalling that domain engineering is a foundation of business components, we point out another key observation in [14] p. 16 (our italics): “Domain engineering is a largely informal, often ad hoc, process. Furthermore, little direct use or reuse is made of domain information in downstream tools. Potentially useful engineering models are often put on the shelf after design and are not employed during software and systems engineering. *Models that could support both informative simulation and proof are inadequately explored.*”

Executability brings component specifications to life through simulation. This imposes the use of a rigorous component modeling language, such as the UML enhanced with action semantics for example. Figure 5 shows a possible way of explicitly formalizing the component intra-computation and intra-concurrency (dashed lines) in UML using Statechart Diagrams. Figure 6 offers an alternative view of the same component specification that limits the understanding on how contained («reside» relationship) sub-components (S1, S2 and S3 in Figure 5) act in specific contexts. In particular, sub-components may be technical components that document what is the required interface of the overall business component. Figure 5 (bottom, right-hand side) gives functioning scenarios that raise the level of information, and thus ultimately confidence. Such animated specification is rather difficult in the model of Figure 6.

The key idea is that the interfaces of business components must be captured based on an event-driven approach. This means that internal operations (w, x, y, z and data acquisition in Figure 5) are delegated to sub-components. In addition, received events should be represented as operations in the interfaces.

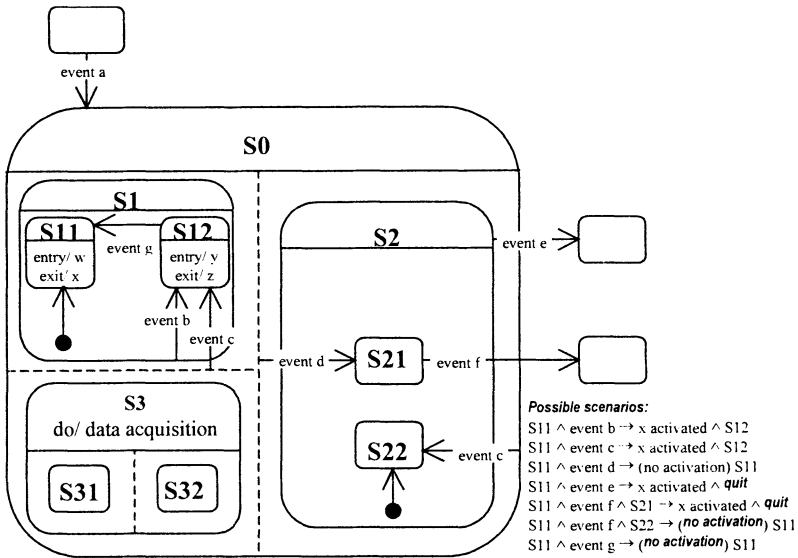


Figure 5. An event/state-driven UML executable component specification

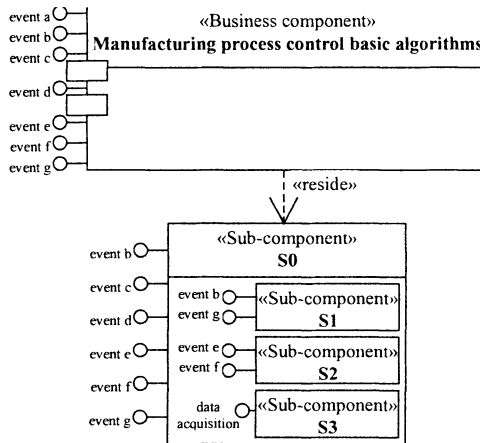


Figure 6. A UML common component specification compacting Figure 5

Coupling our recommendations of how to use UML for business components with a component Architecture Description Language or ADL [22] that supports executability, is an interesting challenge. The combination of executable modular representations such as those in Figure 5 would then be formally defined and based on strict guidelines. Compared to an ADL, UML remains too informal.

3.3 Effective measurable reusability

Business components should have an appearance that is suitable for expressing reusability in general, and composability in particular. Components can be described by means of UML (which is nowadays the most common approach) or by a formal specification language as B, VDM or Z, or by means of implementation-neutral languages such as IDL or XML (eXtended Markup Language). There is, however, no special modeling notation/syntax within component modeling languages to make the component's potential for reuse clear. In other words, connectors which state how reuse is expected to occur are missing. In this respect, the ADLs mentioned above are *more prescriptive than descriptive*, and thus propose constructions that address issues of component combination.

Major challenges relate to the improvement of the formal semantic definition of component relationships. The Kobra method [5], for instance, invented component-specific relationships (composition, clientship, ownership and containment). This also involves the clear characterization of interaction variations induced by each kind of relationship (see Section 3.2 about assembly executability).

Reusability thus becomes realizable based on whether one knows how easily and quickly a component may be integrated into a component framework or an application architecture. Other important issues are the strict isolation of sub-components that essentially play the role of removable parts. Sub-components differ from components in the sense that they are not independently deployable as such. The greatest quality of sub-components is, in fact, that they are interchangeable. From the perspective of the business application layer, sub-components embody a rational splitting of business components. For instance, a monolithic accounting macro-component may awkwardly mix domain-generic and domain-specific sub-components. In this example, an improvement would involve isolating in one component very special calculation demands (business-specific) and in the other the fulfillment of statutory accounting conditions (business-generic).

From the perspective of the underlying technical application layer, sub-components are involved in the required interfaces of business components. Having access to sub-components, and possibly exchanging them with ease, provides the highest degree of composability, and thus flexibility, of the business components in which they are incorporated.

3.4 A seamless transformation for business components within the software development process

Considering the evolution of business components across the software development life cycle, it becomes clear that business components have

multiple facets. From being purely conceptual entities, they are transformed into purely operational entities. To transform models of business components into runtime entities, we have to deal with different formalisms and preferably use canonical component models, such as, for instance, CCM or EJB. Figure 7 shows in UML how an EJB component is organized with a special distinction between Entity Beans and Session Beans as well as their subtypes.

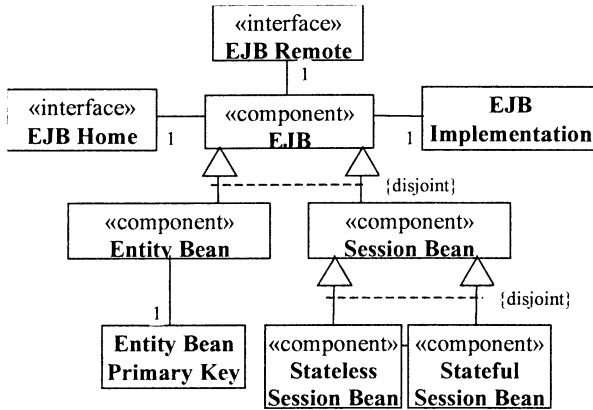


Figure 7. EJB pattern

Figure 8 shows the use of this pattern for a network component in the area of Telecommunication Network Management. Figure 8 sketches a pre-implementation state but supplies no assurance about property preservation from modeling to implementation, even deployment and execution.

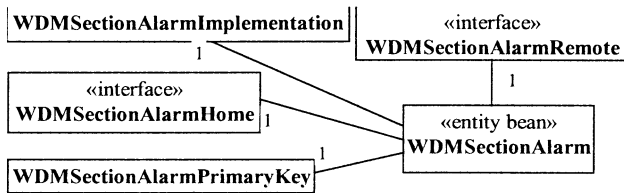


Figure 8. A “Wavelength-Division-Multiplexing Section Alarm” entity bean in the domain of Telecommunication Network Management

The key problem here is the risk of traceability disconnects in the component transformation. We have mentioned an XML variant as a possible expression language for components. Beyond a component modeling language, components have to be stored, retrieved, shared and exchanged in large information systems on the Internet in order to really create a component marketplace. Although IDL, UML, ADLs and even mathematical formalisms can be used to model components, a common background

expression language is needed. Since business components may be available in various forms (even binary forms) that address different development activity activities, it is necessary to manage their evolution/transformation consistently.

4. EXAMPLE: THE OMG CURRENCY BUSINESS COMPONENT

The OMG Currency business component is specified by means of IDL in [21]. Because of the IDL's lack of expressive power, UML Class and Sequence Diagrams are also used to improve its comprehensibility. Curiously, Component and Deployment diagrams are not used although they are suitable for the purpose. We thus take advantage of this example to criticize/enhance the design of component in UML by applying its dedicated component modeling constructions.

“A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.” [6] p. 2-31. Based on this definition, UML provides specific notational elements for Component (Figure 6) and Deployment diagrams. We first present more details on these two kinds of diagrams (known collectively in UML as “implementation” diagrams). We then show how these can be used to improve the UML-based specification of business components such as Currency.

4.1 Notation

A key visual characteristic of a component is the presence of *two small rectangles protruding from the larger rectangle* which symbolizes the component. What is the semantics of these two small rectangles? This is unclear in UML. Thus, components physically contain parts (either drawn inside or linked by means of the «reside» relationship: see Figure 6) that *cannot be executed alone* (i.e. that are dependent software units of execution), and thus *are not independently deployable*. This seems implicit in the UML, but our advice is to distinguish component *parts* by simply avoiding the use of the *two small rectangles protruding from the larger rectangle* (as done in Figure 6 or in Figure 9). This obviously does not prevent component parts from having interfaces and leads to models such as that in Figure 9.

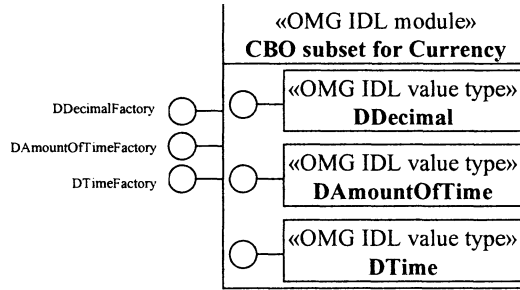


Figure 9. A component part (i.e. an OMG IDL module) itself comprising three parts (i.e. three OMG IDL value types)

In Figure 9, a circle with no name embodies the overall interface of a component part, namely the “DDecimal”, “DAmountOfTime” and “DTime” object types. Alternatively, several circles split the overall interface into logical subinterfaces whose name is for documenting purposes. The “CBO subset for Currency” sub-component (“CBO” meaning “Common Business Object”) is a restricted view of the OMG CBO entire component, and is a unit of deployment in CORBA. Since the Currency business component only uses a small subset, this one is concisely depicted in Figure 9. Note that UML has no systematic mechanisms for providing different views on components.

4.2 Detailed view

A component in the OMG IDL approach is a set of modules, each made up of value types and interfaces. An IDL component specification may thus be viewed as an autonomous software element in the sense that it has to be implemented, it is deployed once and is unique in a component framework. In this respect, the two other modules of Currency are “FbcCurrency” and “CboCurrency” (Figure 10). Logically, Currency in Figure 10 is equipped with two extra small rectangles on the left side, in order to illustrate that it is *not* a part of something but a unit of deployment.

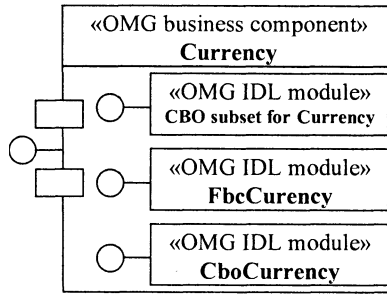


Figure 10. Currency business component general overview

IDL modules organize the overall set of operations supported by a business component, namely the functions that support business logic, in loosely coupled parts. Applying this concept recursively, we arrive at Figure 11 that is the inner part of “FbcCurrency”. We are seriously hindered by the vagueness of the UML dependency relationship (dashed arrow in Figure 11) which does facilitate an explanation of how the object types may interact. The scenario used in [21] pp. 2-23-2-24 partially compensate for the lack of modeling constructs. For instance, any access to a Currency object supposes a Currency book interface, which itself therefore needs to have information on Currency’s clients. The StateIdManager interface in Figure 11 determines identities for clients that are taken into account by Currency. For instance, this is needed in order to cope with client states. By definition, this pattern of functioning cannot be ignored in any interaction. It therefore requires a sophisticated self-contained component formalism that supports, amongst other things, exception description, conditions/constraints on event emission/reception/processing, component state representation. Mixing several UML diagram types is not as satisfactory as having a dedicated language.

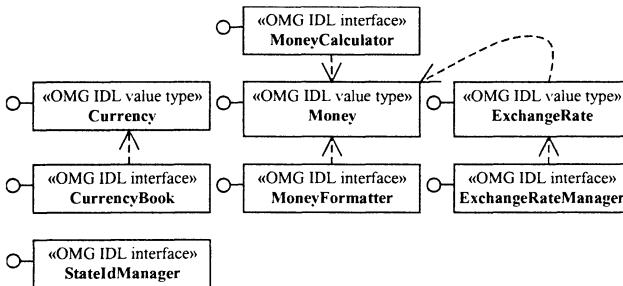


Figure 11. The FbcCurrency sub-component and its intra-coupling

The component formalism in UML is not self-sufficient, and thus is deliberately not used in [21]. Despite these difficulties, the model in Figure 12 is an attempt to synthesize the overall Currency business component. Arrows can end on circles to indicate the use of operations or directly on sub-component rectangles to indicate that they are “used” only for value types. For instance, the Currency interface inside “Cbo Currency” uses the Currency value type inside “Fbc Currency”, itself utilizing “DDecimal” and “DTime” inside “CBO subset for Currency”. As outlined by Heiler in Section 3.1, the exact meanings of these exchanges are difficult to understand.

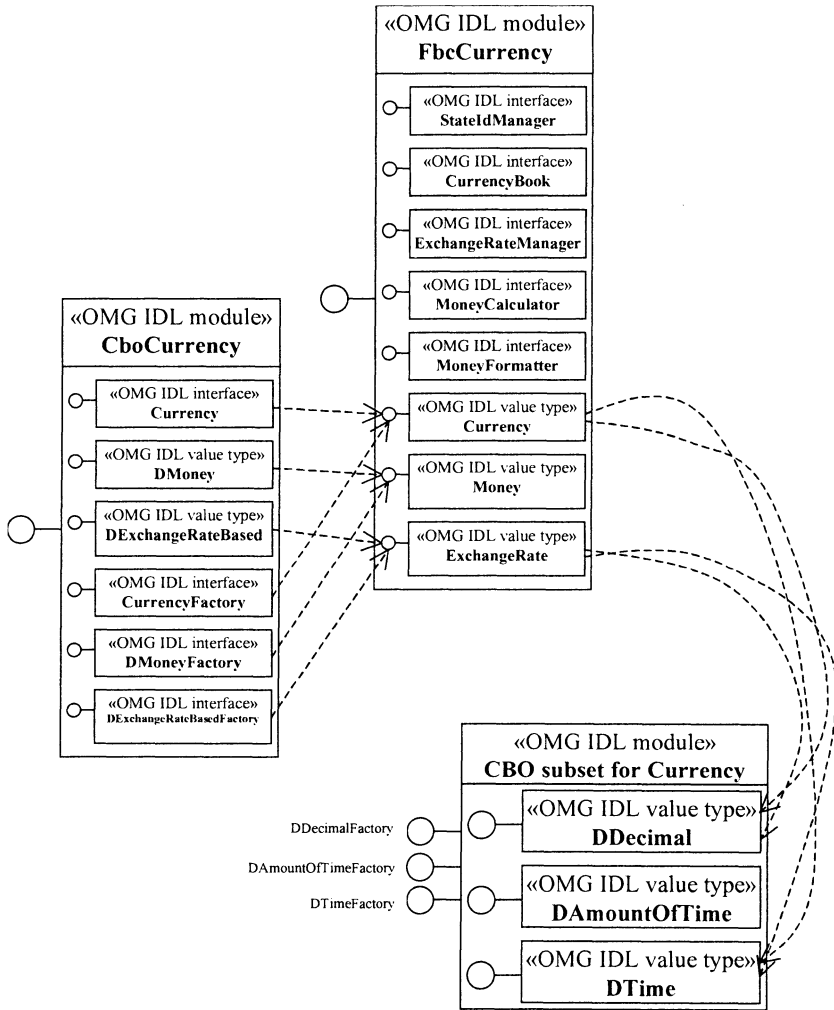


Figure 12. Detailed view of Currency

Intuitiveness of the formalism in Figures 9, 10 and 12 cannot be considered as satisfactory. Namely, schemata are more static than dynamic. The only way to improve understanding is to use additional diagrams whose overlapping with Component Diagrams may be important (Class Diagrams especially). A future step about a true component modeling language in UML is thus required.

5. CONCLUSION

The survey in [2] p. 26 reports that (our italics): “To date, the success of component technology in satisfying the demand for software generated by the IT revolution hinges *largely* on the ability of components to support *domain-specific software reuse*.” This confirms the great importance and expected development of business components in the near future.

In this chapter, one of our main observations is that standardization alone cannot be viewed as the panacea for business components. Certainly, standards create confidence but they are not yet sufficiently stable. It is clear that most the business components existing today came out of the CORBA world (Currency, General Ledger, Air Traffic Control, etc.). However, as long as third-party certification does not occur, it is difficult to ensure that these components or other normalized ones will fulfil conditions linked to legal, security or other demands. Considering the goal for a more open free component market, in this chapter we evaluated the discriminating features that business components have to possess compared to non-business components. Making a distinction between the two types is tricky but nevertheless worthwhile. Our analysis highlights a continuity from business-neutral through business-specific to business-generic components. Business components have thus to offer domain customization capabilities. We therefore recognized the need for configuration interfaces amongst other things help in the substitution of technical components, leading to implementation variations that adapt business components to precise runtime contexts. We also noted the fundamental need for a true component modeling language, beyond IDL or UML that really allows all facets of CBD to be addressed. Semantic interoperability for anticipating, even predicting, assembly behaviors is crucial. Supporting the animation of specification in order to overcome the hesitancy of users in reusing components could be even more helpful in the long term. Finally, supporting and prescribing limited but accurate reuse techniques within a component modeling language must be a priority.

ACKNOWLEDGMENTS

We wish to thank Brian Henderson-Sellers for his (always beneficial) review.

REFERENCES

- [1] C. Szyperski, *Component Software – Beyond Object-Oriented Programming*, Addison-Wesley, 1998.
- [2] L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, K. Wallnau *Volume I: Market Assessment of Component-Based Software Engineering*, Carnegie Mellon University, Software Engineering Institute, TECHNICAL REPORT CMU/SEI-2000-TR-008, ESC-TR-2000-007, May 2000.
- [3] F. Bachman, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, K. Wallnau, *Volume II: Technical Concepts of Component-Based Software Engineering*, Carnegie Mellon University, Software Engineering Institute, TECHNICAL REPORT CMU/SEI-2000-TR-008, ESC-TR-2000-007, May 2000.
- [4] D. D'Souza, A. Cameron Wills, *Objects, Component and Frameworks with UML, The Catalysis Approach*, Addison-Wesley, 1999.
- [5] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Peach, J. Wust, J. Zettel, *Component-Based Product Line Engineering with UML*, Addison-Wesley, 2002.
- [6] Object Management Group, *OMG Unified Modeling Language Specification*, Version 1.4, September 2001.
- [7] J. Cheesman, J. Daniels, *UML Components: A Simple Process for Specifying Component-Based Software*, Addison-Wesley, 2001.
- [8] G. Heineman, W. Council, *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, 2001.
- [9] O. Sims, *Business Objects – Delivering Cooperative Objects for Client-Server*, McGraw-Hill, 1994.
- [10] S. Wartik, R. Prieto-Diaz, *Criteria for Comparing Reuse-Oriented Domain Analysis Approaches*, International Journal of Software Engineering and Knowledge Engineering, 2(3), pp. 403-431, 1992.
- [11] Object Management Group, *General Ledger Specification*, Version 1.0, February 2001.
- [12] J. Roschelle, C. DiGiano, M. Koutlis, A. Repenning, J. Phillips, N. Jackiw, D. Suthers, *Developing Educational Software Components*, IEEE Computer, IEEE Computer Society Press, September 1999.
- [13] B. Meyer, *Reusable Software – The Base Object-Oriented Component Libraries*, Prentice Hall, 1994.
- [14] National Coordination Office for Information Technology Research and Development, *HIGH CONFIDENCE SOFTWARE AND SYSTEMS RESEARCH NEEDS*, January 2001.
- [15] H. Kilov, *Business Specifications – The Key to Successful Software Engineering*, Prentice Hall, 1999.
- [16] M. Fowler, *Analysis Patterns – Reusable Object Models*, Addison-Wesley, 1997.
- [17] Object Management Group, *Time Service Specification*, Version 1.0, May 2000.
- [18] Object Management Group, *Air Traffic Control Specification*, Version 1.0, May 2000.
- [19] S. Heiler, *Semantic Interoperability*, ACM Computing Surveys, ACM Press, 27(2), pp. 271-279, 1995.

- [20] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, Second Edition, 1997.
- [21] Object Management Group, *UML Specification*, Version 1.0, June 2000.
- [22] N. Medvidovic, R. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Transactions on Software Engineering, IEEE Computer Society Press, 26(1), pp. 70-93, 2000.

Chapter 2

Model Driven, Component-Based Development

Colin Atkinson and Hans-Gerhard Groß

IESE, Germany

Abstract: As complex behavioral abstractions, business components in many ways can be viewed as (sub)systems in their own right, and thus need to be treated as such. In particular, techniques are needed that enable the properties, requirements and architectures of business components to be modeled in the early analysis and design phases of software development. UML component diagrams are useful for describing the physical components from which executing systems can be deployed but are totally inadequate for describing the rich behavior and relationships of business components. This chapter addresses this need by describing a practical, systematic technique for modeling business components and the systems assembled from them. The chapter starts by describing the basic modeling artifacts used to model components, and illustrates their applications in the context of a small case study. It then goes on to discuss advanced modeling concepts needed to support the hierarchical nesting of components.

Key words: Component modeling, model-driven architecture (MDA), component-based development, analysis, design

1. INTRODUCTION

As the key logical building blocks from which software systems will be assembled, business components are of importance right from the early requirements analysis and design phases of the software life-cycle. However, the current generation of component technologies, such as EJB/J2EE, .NET and CORBA, focus primarily on the implementation and deployment phases of development, and provide little support for components earlier in the development process. With the continued uptake of the UML [1], and the OMG's emphasis on the Model Driven Architecture [2], providing practical support for the development and application of business components implies

the need for a systematic technique for using the UML to model their properties, relationships and interactions.

At first sight it might appear that UML component and deployment diagrams are suitable for this purpose, since their stated goal is to model component types and their deployment on a network of computers. However, this is far from being the case. UML component and implementation diagrams are very much aimed at capturing the implementation-level properties of physical components such as EJBs and .NET components, and lack the expressive power needed to capture the rich behavior of business components. In fact, large business components can essentially be regarded as systems in their own right with all the associated complexity of behavior and functionality. Using the UML to model such components therefore involves the deployment of multiple, intertwined diagram types each providing a view onto a different aspect of the component's characteristics.

In the last few years several new methods have been published to address this problem by supporting the analysis and design of component-based systems. The most prominent of these include the SELECT Perspective [3], Catalysis [4] and UML Components [5]. However, while these methods have undoubtedly made a significant contribution to the state of the art, they focus primarily on the *specification* of business components, and pay relatively little attention to the UML-based representation of their realizations. Moreover, although they all recognize the value of the hierarchic nesting of components, they give little insight into how this should be achieved in terms of inter-related UML models. This is a problem because organizing a system in terms of recursively nested components is one of the key concepts distinguishing the component paradigm from other development approaches. It is also an important theme in the OMG's Enterprise Distributed Object Computing (EDOC) [6] profile which standardizes business component modeling ideas.

As well as supporting the specification of components, therefore, a comprehensive method for documenting component-based, model driven architectures must also provide an approach for modeling their architectures/designs, and for organizing all the models of a component hierarchically to reflect the system's composition structure. A recent method developed to address precisely this need is the Kobra method [7]. This builds on the techniques developed in the aforementioned methods by adding support for the description of component realizations and the recursive nesting of components. In this chapter we describe Kobra's approach for modeling components and component architectures in terms of the UML, with a special emphasis on recursive nesting of components and the models that describe them.

2. MODELING BUSINESS COMPONENTS

When addressing the problem of modeling business components it is important to accommodate the fact that components manifest themselves both at run-time and at development time. At run-time, components constitute the objects from which the overall functionality of the system is constructed, while at development time components correspond to types which describe the properties of the component instance existing at run-time. The overall concept of a component therefore has both a type and an instance facet. Like the UML, Kobra uses the unqualified phrase “component” to refer to component types.

When discussing the nesting of components it is important to be clear whether it is the nesting of run-time component instances or development time components which is meant. This distinction is illustrated clearly in Figure 1 which introduces the example that we will use throughout the remainder of the chapter. The left hand side of this figure illustrates a tree of components making up a small banking application which we refer to as the Simple Banking System, or SIB for short. Using the UML “black-diamond” symbol to represent composition, this illustrates that a *Bank* component instance, is composed of two sub-component instances, one instance of the component *Teller* and another instance of the component *Converter*. Furthermore the *Teller* component instance is itself composed of a *LookUpTable* component instance.

The right hand side of Figure 1 illustrates the types from which these component instances are instantiated. Obviously the component instance *:Bank* is an instance of the component *Bank* and so on. The organization of the components on the right hand side of Figure 1 resembles that of the component instances on the left hand side, but it describes the component containment hierarchy rather than the component instance composition hierarchy. To understand the difference it is important to recognize that components (i.e. component types) also have two distinct facets: a class facet, which describes the properties and potential relationships of their instances, and a module facet, which describes the properties of the component from the point of view of a container. In Kobra, a component (viewed as a module) contains another component (viewed as a module) when the definition of the second is contained within the definition of the first. In UML terms, the module facet of a component corresponds to the concept of a package. Thus, the right hand side of Figure 1 indicates that the *Bank* component contains the *Teller* and *Converter* components, and the *Teller* component contains the *LookUpTable* component. The role of the *BankContext* will be explained shortly in section 3.1.

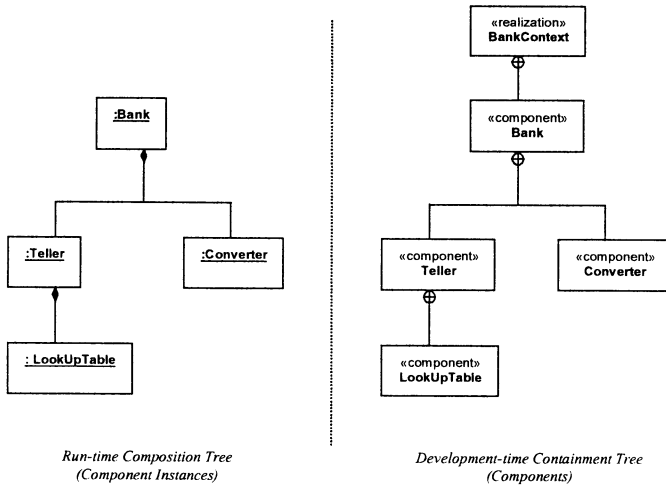


Figure 1. SIB System - Version 1

The run-time composition tree and development time containment tree do not always have to have the same shape. However, there are strict visibility rules which determine what composition trees can be derived from a given containment tree. These are the visibility rules usually associated with nested modules or packages in a modeling or programming language. The basic idea is that a module has visibility of everything in the private parts of every modules which contain it (directly or indirectly) unless that thing happens to be another module. In this case the first module only has visibility of that module's (i.e. the second module's) public part. Thus, in the containment tree on the right side of Figure 1, the component *LookUpTable* can see anything defined in all the components that contain it, in this case *Teller* and *Bank*. This set of things includes *Converter*, since this is defined in *Bank*. However, because *Converter* is a component, *LookUpTable* can only see the public part of *Converter*. It does not have visibility of anything defined in the private part of *Converter*, including any of its subcomponents if there happened to be any. We explain later what we mean by the public and private part of a component.

2.1 Specification

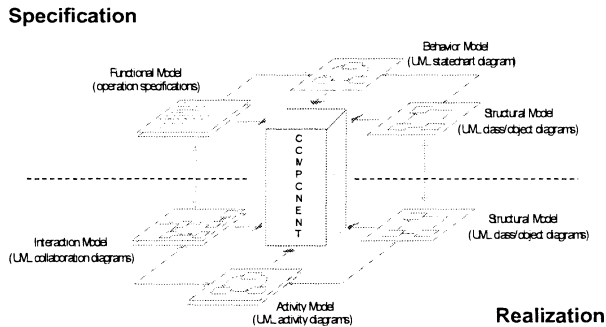


Figure 2. Component Modeling Artifacts

Figure 2 illustrates the set of primary modeling artifacts which are used to describe the properties of a business component in Kobra. The specification of a component captures all the externally (i.e. client) visible properties of a component, and thus when viewing the component as a class, the specification represents the component's interface. When the component is viewed as a module, however, the specification represents the public part of the component since it contains everything that is meant to be externally visible. Finally, from the perspective of the development process the specification defines the requirements that realizations of the component have to satisfy.

As illustrated in Figure 2, a component specification consists of three tightly interwoven models. These models, which are derived from those popularized in the classic OMT method [8], provide inter-related views on distinct aspects of the externally visible characteristics of the component. The structural model forms the foundation for the rest of the specification by defining the externally visible concepts manipulated by the component, the externally visible components with which the component interacts, and any externally visible structure of the component. An example of a component's specification structural model is shown in Figure 3.

Although this is quite a simple example, this diagram contains a lot of information about the properties of the *Bank* component. The «*subject*» stereotype serves to identify which component the diagram is focused upon – in this case *Bank*. From the diagram it can be seen that the *Bank* component has one logical attribute which stores the number of accounts held by the *Bank*, and eight operations which provide various account manipulation and currency conversion services. The diagram also shows the assumed nature of accounts in terms of the *Account* class. In particular, it shows that a *Bank* manipulates multiple accounts, each of which has four distinct logical

attributes including a *limit* attribute to indicate how far a customer can overdraw an account.

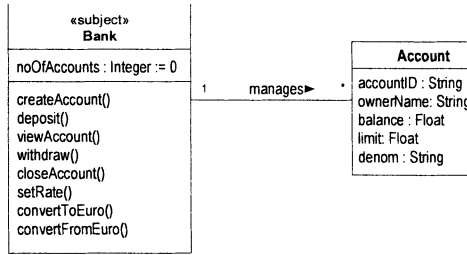


Figure 3. Bank Specification Structural Model

The behavioral model describes the logical states exhibited by the component. This includes a description of the events which stimulate the component and the circumstances (i.e. states) under which the operations of the component may execute. The behavioral model is essentially a state machine which may be represented in a tabular form or in a graphical form as a UML state chart diagram as illustrated in Figure 4. This diagram shows the logical states exhibited by the bank. Initially a new *Bank* instance is in an *Empty* state when it has neither accounts nor exchange rates stored. If the *createAccount()* operation is the first to be executed from this state the *Bank* instance enters a state where only account manipulation operations (except *setRate()*) are permitted and then only in the default currency of Euros. Alternatively, if the *setRate()* operation is the first to be executed in the *Empty* state the *Bank* instance enters a state in which currency conversion operations are permitted but no account manipulation operations (except *createAccount()*). Only when the *Bank* instance has both accounts and exchange rates stored are the full range of services available.

The functional model describes the logical effects of the component's operations in terms of the information in the structural and behavioral models. It consists of a collection of operation specifications, one for each operations supported by the component. As illustrated in Figure 5, which shows the specification for the *withdraw()* operation, an operation specification is a textual table consisting of a sequence of so called "clauses". The most important of these clauses are the *assumes* clause and the *results* clauses. The former defines the preconditions which must be true for the operation to be sure to function correctly, while the latter defines the postconditions which become true if the operation is executed when the precondition is true. The other clauses summarize information from the *assumes* and *result* clauses. Specifically, the *receives* clauses lists the parameters received by the operation, the *returns* clauses list information returned by the operation to the invoker, the *changes* clause lists items from the *Bank*'s data model which are

affected by the operation and the *rules* clause identifies any logical definitions used in the specification.

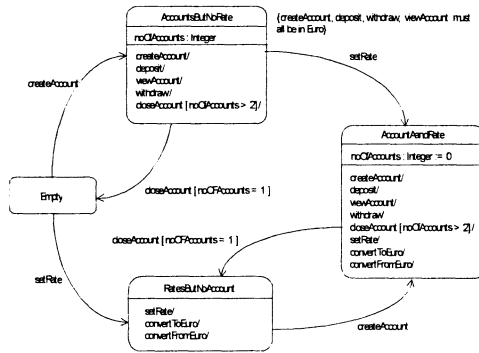


Figure 4. Bank Behavioral Model

As illustrated in Figure 5, these clauses can be written in a natural language style, or they can be written in a more formal style such as, for example, in OCL. In whatever form they are represented, however, the important point is that the specification must only refer to concepts in the component’s specification structural and behavioral models. This not only ensures the mutual consistency of the models, but also provides a criterion for establishing how much information they need to contain.

Name	<i>withdraw</i>
Informal Description	an amount of money in a particular currency is withdrawn from an account
Receives	ID : String currency: String amount: Float
Returns	a Boolean indicating whether the withdrawal was possible
Changes	account with accountID = ID
Rules	if currency = account denom then equivalentAmount = amount else equivalentAmount = amount converted to denomination
Assumes	ID is a valid account ID currency is a valid currency identifier
Result	if account.balance + account.limit ≥ equivalentAmount account.balance := account.balance - equivalentAmount and <i>True</i> has been returned else <i>False</i> has been returned.

Figure 5. “withdraw” Operation Specification

2.2 Realization

The realization of a component describes how the component fulfills its obligations by interacting with other components and objects. As such it represents the private architecture and/or design of the component. This

applies the well established software engineering principle that the description of how a software artifact works should be hidden from users of the artifact in order to facilitate change. The realization also contains any private subcomponents which the component defines for its own realization. Whereas the specification represents the public part of the component, therefore, the realization represents the private part which is not normally visible to external components.

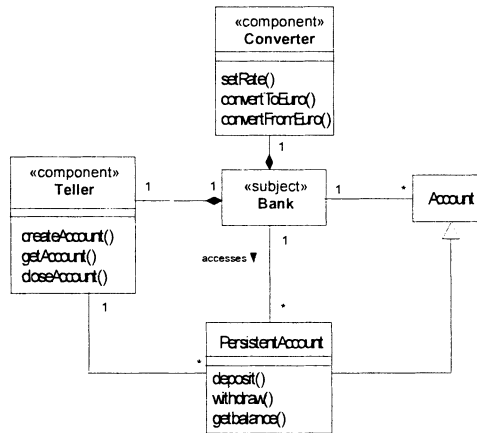


Figure 6. Bank Realization Structural Model

As illustrated in Figure 2, like a specification a realization consists of three primary models. The structural model describes the components and types involved in the realization of the component in questions. Figure 6 illustrates the realization structural model for the *Bank* component. In accordance with the development time containment tree (illustrated in Figure 1), this indicates that two private subcomponents, *Teller* and *Converter*, are used by *Bank* to realize its services. It also indicates that a more specialized form of *Account*, known as *PersistentAccount*, is used in the realization. The realization structural model of a component is a refinement of its specification structural model. Thus, the information contained in Figure 3 is implicit in Figure 6 and does not need to be explicitly reproduced. If desired, the type information in the class diagram (Figure 6) can be augmented by an object diagram to depict the configuration of the component instances within the realization.

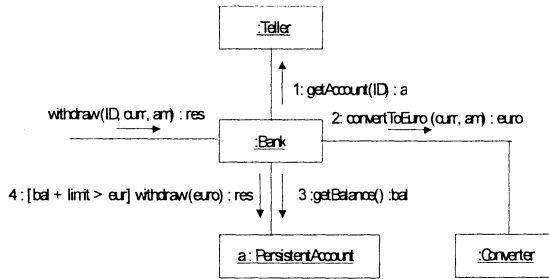


Figure 7. “withdraw” Collaboration Diagram

The structural model identifies the components belonging to the realization in hand. This static information must be augmented by dynamic information which defines how the component instances interact to realize the required services. This is the role of the other two models, the interaction model and the activity model.

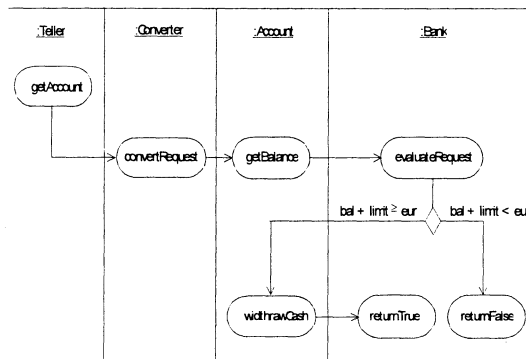


Figure 8. “withdraw” Activity Diagram

The interaction model is composed of a set of UML interaction diagrams (i.e. sequence or collaboration diagrams) each of which describes how the component and object instances interact to realize a particular service. Figure 7 illustrates the collaboration diagram which describes the realization of the *Bank* component’s *withdraw()* operation. Naturally, only instances of components in the realization structural model can appear in such a diagram, and messages can only be sent to these instances if they are listed as operations in the model. Also, it is essential that the collaboration diagram fulfills the requirements defined in the operation’s specification, since this defines the desired effects of the operation.

The activity model essentially documents the same information as the interaction model but from a different perspective. Whereas the interaction model emphasizes object interactions, the activity model emphasizes control

flow. It does this by documenting the algorithms used to realize the services in terms of activity diagrams. Figure 8 shows such an activity diagram for the *withdraw()* operation of the *Bank*. If a collaboration diagram and an activity diagram are both created for a given operation they must naturally agree on the form of the algorithm used. As in the interaction model, there is one diagram for each distinct activity supported by the operation.

3. NESTED COMPONENT ARCHITECTURES

Up to this point we have described how the characteristics and realization of a single business component can be modeled using a suite of inter-related UML diagrams, each providing a view on a different aspect of the component and its services. In this section we explain how this approach can be generalized to support the nesting of components, and thus to support a comprehensive and recursive approach to component-based development.

As explained in the previous sections, there are two distinct dimensions involved in the hierarchic organization of components: the run-time dimension related to the nesting of component instances and the development time dimension related to the nesting of components (i.e. component types). Following the terminology of the UML, the run-time nesting of component instances is referred to as composition, while the development-time nesting of components is referred to as containment. The basic principles involved in creating each hierarchy individually are fairly straightforward — the real challenge is to define how the two dimensions should relate to one another.

The basic ideas involved in nesting components at development time are illustrated in Figure 9 and depend on the module-like properties of components. Assuming that the models/diagrams for a component are contained in the module (in UML terminology — package) for that component, the essential idea is to embed the module for a subcomponent within the module of its parent component. As illustrated in Figure 9, this implies that the models for a subcomponent are subservient to (i.e. based upon) those of the parent component. In particular, the nature of a component's specification is determined by the nature of the realizations of the component's parent and clients. Thus, in the case of the SIB example, the specification of *Teller* is derived from of the realization of its parent component (i.e. *Bank*) since this is its only client. Figure 9 shows the same hierarchy of components as that depicted on the right hand side of Figure 1, but with an emphasis on the models used to describe the components.

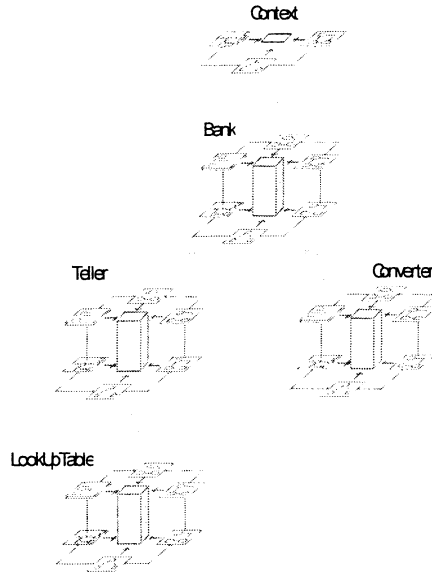


Figure 9. Models in a Containment Hierarchy

In KobrA, therefore, all components in a component-based system are treated in the same way and are modeled using the same general suite of UML models. To illustrate this point further, Figure 10 shows the specification structural model of the *Teller* component, one of the subcomponents of the *Bank* component. This diagram provides a structural view of the *Teller* as an independent component, but is nevertheless strongly related to the information in the *Teller's* parent (i.e. *Bank*). In this particular case, since *Teller* has no other clients than *Bank*, the information presented in *Teller's* specification structural model is essentially the subset of information in the *Bank's* realization structural model which has a bearing on the properties of *Teller*.

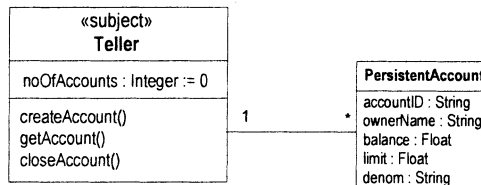


Figure 10. Teller Specification Structural Model

Naturally the other specification models of *Teller* are also driven by the nature of the *Bank's* realization in a similar way.

The composition hierarchy captures the run-time manifestation of the nesting of components as it relates to component instances. In the development time description of a component-based system, this information is captured in the structural model of component realizations. For example, the black composition diamonds in the Bank's realization structural model (Figure 6) explicitly indicate that an instance of *Bank* is composed of an instance of *Teller* and instance of *Converter*. The fact that a *Teller* instance is itself composed of a *LookUpTable* is not defined at the level of the *Bank* since this is a matter for the private realization of *Teller*. Thus, this information would be documented in the *Teller's* realization.

3.1 Context Realization

Component realizations play an instrumental role in enabling both the run-time composition tree and the development time containment tree to be modeled in a clean and straightforward way. The introduction of realizations as an equal partner to specifications in documenting the properties of components is one of the main contributions of Kobra. However, the importance of realizations does not stop there. They also have an important role to play in capturing the environment in which business components are expected to operate.

As explained in the previous subsection, component specifications are developed in the context of the realization of the component's parent (and/or other clients of the component) since this defines what the component is expected to do. But this begs the question as to what happens at the top of the composition hierarchy where the component representing the entire system needs to be specified. The answer is depicted in Figure 9. In Kobra the environment of the system component is itself documented as a realization. For obvious reasons this is known as the context realization since it describes the properties of the context of the system component.

Interestingly, the models contained within a realization closely resemble those that are generated in traditional UML-based requirements elicitation and analysis techniques. Operation (or activity) specifications, interaction diagrams and class diagrams are the essential products of use case analysis, and activity diagrams are the essential products of business process modeling. This close correspondence between Kobra realization models and traditional analysis models might at first seem surprising, but when one considers the fact that introducing a new system into a business environment always changes that environment, and thus can be viewed as a form of realization, the relationship between analysis and realization is obvious.

Since they model the emergent properties of a collection of components, Kobra realizations actually have an analog in most contemporary component-oriented development methods. Other names used to capture this idea include

ensemble [9], community or assembly. The only difference in Kobra is that the concept is reapplied recursively in the body of the system's architecture to describe the designs of internal components as well as to describe the properties of the system and/or its environment.

3.2 Tree Alignment

The previous subsections have discussed the run-time composition tree and the development time containment tree as separate dimension of a component-based architecture, but did not discuss how they relate to one another. The whole point of the Kobra approach is to provide heuristics and strategies for *aligning* the two trees so that the development of components can be driven by the required run-time architecture.

Using composition as the basis for aligning the run-time and development dimensions of a component-based architecture is difficult because a system may in fact not always contain a single composition tree. Composition, as defined in the UML, has a number of strict requirements (such as lifetime dependency of the part component on the composite component) which may not always be met. Thus, a system may have multiple disjoint composition trees, or it may not have a composition tree at all. One run-time structure which all component-based systems do have, however, is a creation tree. This is derived from the creation relationship that exists when one component instance creates another. Since every component instance must have exactly one creator, the creation relationships in a system always traces out a tree that is rooted at the component instance representing the system. Moreover, since composition is closely related to creation, the creation tree subsumes any composition trees within the system, at least at the point when it is first generated. For these reasons it is the creation tree which is actually used as the basis for the alignment of the run-time and development time dimensions.

The basic requirement when aligning the two trees is that the shape of the containment tree must allow the required creation tree to be generated in accordance with the component visibility rules described previously. For a given creation tree there are usually numerous compatible containment tree shapes that could be used. In fact, a universally applicable shape is a two level tree which has the system component as its root, and all other components as its direct subcomponents. By definition, this shape gives all components visibility of all other components, and thus allows any form of creation tree to be created. However, such a containment tree wastes the opportunities to improve the cohesion and coupling of the architecture by hiding private realization details. The goal therefore is to find the optimal balance between visibility and information hiding based on the needs of the desired run-time creation tree.

The various principles and mechanisms that can be used to achieve this balance are the topic of the following sections. In each section we address a particular principle and/or mechanism and show how it can be used to determine the alignment of the creation and containment trees. In fact, the principles and mechanisms are very similar to those used by programmers working in traditional object-oriented programming languages. The only difference is that in Kobra these are applied to models at the level of business objects, while with programming languages they are applied at the level of modules, classes and objects. To provide a concrete illustration of the analogy between tree alignment in Kobra and tree alignment in an object-oriented program, in each of the following sections we present the outline of a “reference” Java program which has the same run-time and development time architecture as the Kobra models. To mirror the visibility rules of component containment we use the concept of inner classes in Java, since these have the same visibility semantics. However, packages could also have been used.

```
public class BankContext {
    public class Account {};
    public class Bank {
        private class Teller {
            private class LookUpTable {};
            private LookUpTable L = new LookUpTable();
        };
        private class Converter {};
        private Teller T = new Teller ();
        private Converter C = new Converter();
    };
    public Bank B = new Bank ();
};
```

Figure 11. SIB Example Version 1 Reference Implementation

Figure 11 illustrates the reference implementation for the architecture shown in Figure 1. The containment hierarchy is captured by the text in regular font while the creation tree is captured by the italicized text. Indentation indicates the scoping (or embedding) level of an artifact. Thus from Figure 11 it can be seen that that class *BankContext* contains the definition of two classes, *Account* and *Bank*. *Account* does not represent a component, but is needed as a supporting data structure. The class *BankContext* also creates an instances of the system component *Bank*. The *Bank* class, in turn contains the definition of two inner classes, *Teller* and *Converter*. *Bank* also creates an instance of each of these classes. Finally, the class *Teller* defines an inner class *LookUpStore*, and creates one instance of it. Thus, the organization of the class definitions in Figure 11 exactly matches the containment tree on the right hand side of Figure 1, and the organization of the creation of objects exactly matches the creation tree (in this case also composition tree) on the left hand side of Figure 1.

It is important to note that while this design could be (and has been) used to create working implementations of the system, it is not meant to imply that

a real implementation would take this form. Real implementations of this form of architectures would need to take into account many real world performance and quality issues which are not addressed here.

4. COMPONENT SHARING

One of the most important influences on the shape of the containment tree is the need for two or more component instances to have visibility of the same component type. This need arises for example when one component instance needs to use the services of another component instance that it did not create, or when two components simply wish to have their own private copy of a given component (type). The second situation is illustrated in Figure 12.

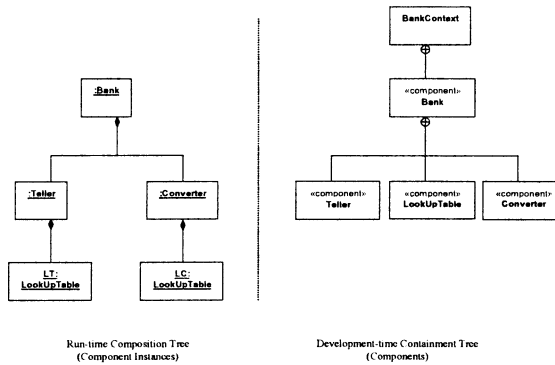


Figure 12. SIB Example - Version 2

In this version of the SIB system architecture, the *Converter* component instance uses its own private *LookUpTable* to store exchange rates. This means that *Teller* and *Converter* both need to have visibility of *LookUpTable*, otherwise they could not both instantiate it. With the mechanisms currently at our disposal the only way to achieve this is to elevate *LookUpTable* from being a private subcomponent of *Teller* to being a peer of both *Teller* and *Converter*, as illustrated in Figure 12. This gives both components the necessary visibility of *Teller*. The corresponding references implementation is shown in Figure 13.

```
public class BankContext {
    public class Account {};
    public class Bank {
        private class LookUpTable {};
        private class Teller {
            private LookUpTable LT = new LookUpTable ();
        };
        private class Converter {
```

```

        private LookUpTable LC = new LookUpTable ();
    };
    private Teller T = new Teller ();
    private Converter C = new Converter ();
};
private Bank B = new Bank ();
};

```

Figure 13. SIB Example Version 2 Reference Implementation

5. PUBLIC CONTAINMENT

Because the need for components to share visibility of other components is very common, using the aforementioned technique to ensure the required visibility often results in a flat tree where almost all the components are direct children of the root. This not only increases the coupling of the components in the architecture, but also destroys any resemblance between the creation and containment trees.

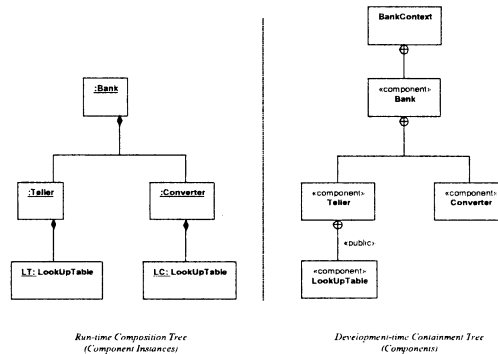


Figure 14. SIB Example - Version 3

To alleviate this problem, Kobra supports the concept of public containment in which a component can declare a subcomponent to be publicly visible. This means that when a component has visibility of another component, it also has visibility of that component's public subcomponents. As illustrated in Figure 14, public containment allows a component to remain encapsulated by the parent component to which it most naturally belongs and also enables the shape of the containment to much more closely resemble the creation tree.

The run-time architecture of version 3 of the SIB system is identical to that of version 2. The only difference is that the containment tree uses public containment to make *LookUpTable* visible to *Converter*. This allows *LookUpTable* to remain a subcomponent of *Teller* whilst at the same time

being visible to *Converter*. In the UML models, the fact that a component has a public subcomponent is shown by including it in the specification structural model as well as in the realization structural model, and also by annotating the relation between them using the «public» stereotype. In the reference implementation, the fact that *LookUpTable* is a public subcomponent of *Teller* corresponds to making it a public inner class, as illustrated in Figure 15.

```
public class BankContext {
    public class Account {};
    public class Bank {
        private class Teller {
            public class LookUpTable {};
            private LookUpTable LT = new LookUpTable ();
        };
        private class Converter {
            private LookUpTable LC = new LookUpTable ();
        };
        private Teller T = new Teller ();
        private Converter C = new Converter ();
    };
    private Bank B = new Bank ();
};
```

Figure 15. SIB Example Version 3 Reference Implementation

6. PUBLIC COMPOSITION

Although it is usually best to hide the parts of a composite component instance so that they cannot be seen by its clients, this is not always the case. Sometimes the fact that a component instance is composed of multiple parts is a logical and natural aspect of the component's definition. For example, a car has many parts (i.e. components) which are hidden to its users (e.g. the engine and gear box) but it also has many parts which are visible (e.g. the steering wheel and gear stick) and which contribute towards the users perception of what a car is. The same situation also exists frequently in the case of software components. For example, user interface components often make their subcomponents directly visible to their users. Forcing such logically public parts to be conceived of as private introduce a lot of additional modeling overhead because the services which they provide must be re-exported by the composite object. At the level of the composite object these services (i.e. operations) essentially play the role of forwarding operations and thus serve no proper role.

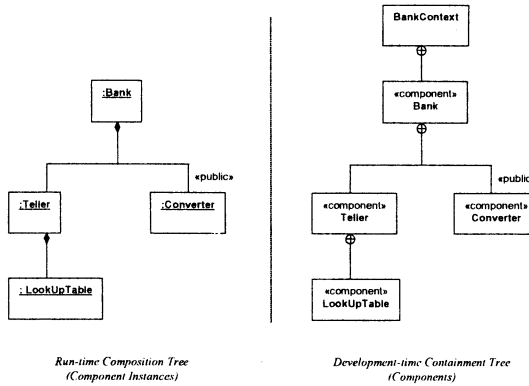


Figure 16. SIB Example - Version 4

To allow such a situation to be modeled in a more natural way Kobra also supports the concept of public composition. This is the analog of public containment, but at run-time rather than development time. As might be expected the two concepts are complementary, and it is common to use public containment to attain the component (type) visibility needed to public composition.

```

public class BankContext {
    public class Account {};
    public class Bank {
        private class Teller {
            private class LookUpTable {};
            private L = new LookUpTable ();
        };
        public class Converter {};
        public Teller T = new Teller ();
        public Converter C = new Converter ();
    };
    public Bank B = new Bank ();
};

```

Figure 17. SIB Example Version 4 Reference Implementation

This is the situation illustrated in Figure 16. The *Converter* instance is defined to be a public part of *Bank* in the composition tree, so the *Converter* component is defined to be a public subcomponent of *Bank* in the containment tree. This gives clients of the *Bank* the visibility needed to access the *Converter* instance directly. The advantage of this architecture is that *Bank* is now relieved of the responsibility of having to support services actually implemented by *Converter*.

7. COMPONENT SPECIALIZATION

Using public composition to make a part visible to clients of a composite component simplifies the modeling of such scenarios, but it forces all of the part's interface to be exposed. Often, it would be better to make only some of its interface public, and keep the rest private. In Kobra, this can be achieved by a similar mechanism to that found in object-oriented programming languages such as Java. The basic idea is that independent component specifications can be defined (without associated realizations) to play the role of abstract classes (or interfaces in Java). Clients of such a specification use them just as if they were a normal component, but when a community of components is assembled within a realization an instance of a suitable specialization component must be supplied.

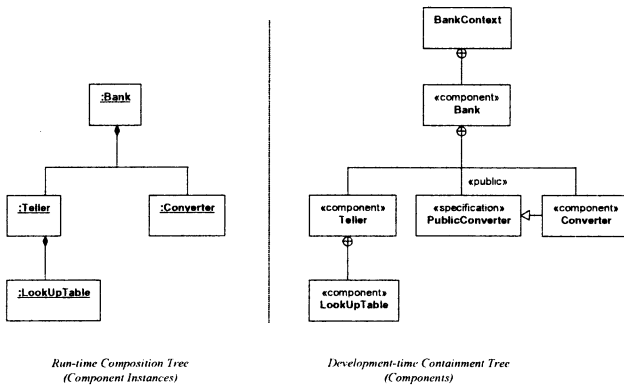


Figure 18. SIB Example - Version 5

Using this mechanism it is possible to control how much of a private part's interface is actually exported, as illustrated in Figure 18. *PublicConverter* is a specification which defines the set of services that a *Bank* instance wishes to export. It therefore plays the role of an abstract class (or in Java an interface) in an object-oriented language. The full *Converter* component is then defined as a specialization of this specification, and adds the extra features that *Bank* does not wish to export. In Java this corresponds to a class implementing an interface as illustrated in Figure 19.

```
public class BankContext {
    public class Account {};
    public class Bank {
        public interface PublicConverter
        private class Teller {
            private class LookUpTable {};
            new LookUpTable L;
        };
        private class Converter implements PublicConverter{};
    };
};
```

```

        private Teller T = new Teller ();
        public Converter C = new Converter ();
    };
    public Bank B = new Bank ();
};

```

Figure 19. SIB Example Version 5 Reference Implementation

8. DISTRIBUTED COMPONENTS

The previous versions of the SIB example have all assumed that a single component, *Bank*, subsumes the functionality of the entire application, and thus in essence represents the system to be developed. While this represents an effective architecture for component-based systems, it is not the only way of deploying components. Many applications are actually organized as a set of distributed component instances, each executing on its own processor (or node as it is called in the UML). Such applications are therefore composed of several independently deployable components which have to be independently instantiated and given access to each other dynamically. This model of system configuration has assumed greater importance since the advent of Web services. Web services are essentially independently created component instances which can be accessed dynamically via the Internet.

Figure 20 illustrates how this scenario is handled in KobrA. Since there is no longer an explicit component that represents the root of the containment and creation trees, the context has to play this role. The containment tree on the right hand side of Figure 20 illustrates that the *Converter* and *LookUpTable* components have been elevated to the same level as *Bank* and thus essentially represent “systems” in their own right. The «acquires» relationships between the components also highlights the fact that instances of these components have to gain access to their servers dynamically.

The right hand side of Figure 20 illustrates a particular configuration of component instances generated from these components. The *BankContext* is included in this tree as a “pseudo” component instance to serve as the root of the creation tree, although no actual *BankContext* exists. The component instances shown as parts of the *BankContext* have to be individually created and initialized by some means not specified by the model (e.g. manually). Also, note again the use of the «acquires» relationships to indicate that components have to acquire access to their servers dynamically.

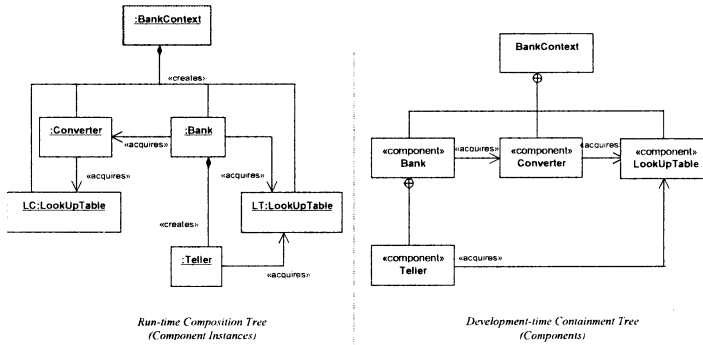


Figure 20. SIB Example - Version 6

The reference implementation for the distributed version of the SIB example simply elevates the independently executable components, and their creation, to the level of the *BankContext*. *Teller* is now the only component instance which is regarded as being part of another component, and is automatically generated by that component.

```
public class BankContext {
    public class Account {};
    public class LookUpTable {};
    public class Converter {};
    public class Bank {
        private class Teller {
            private Teller T = new Teller ();
        };
        public Bank B = new Bank ();
        public LookUpTable LT = new LookUpTable ();
        public LookUpTable LC = new LookUpTable ();
        public Converter C = new Converter ();
    };
};
```

Figure 21. SIB Example Version 6 Reference Implementation

9. CONCLUSION

The advent of component-based development in general, and business components in particular, promises to radically improve the way in which software is developed and maintained. However the effective description and deployment of business components is contingent upon the availability of a simple and systematic method for modeling their properties. The required approach must not only provide a way of specifying the interfaces of components, but must also provide a means to model the way in which they are realized. Moreover, the approach must allow these models to be organized hierarchically so that the advantages of component assembly can be applied at all levels of granularity in a recursive manner.

In this chapter we have described a method, known as Kobra, which meets this needs in terms of the UML. The method's key distinguishing features are its introduction of component realizations to model how the services of a component are realized in terms of a community of other components, and the recursive application of UML models to describe the emergent properties of a nested hierarchy of components. The method also defines strict rules as to how these UML models should be applied and how they should be related to one another.

To enable the development-time containment tree to be optimally aligned with the run-time creation tree, the basic component visibility rules are augmented in Kobra by several advanced modeling concepts, including public composition, public containment and component specialization. Together with the basic component development techniques these enable the properties of a community of business objects to be modeled in a straightforward yet comprehensive manner, and the true potential of model driven, component-based development to be fulfilled.

REFERENCES

- [1] Object Management Group, *OMG Unified Modeling Language Specification*, Version 1.4, September 2001.
- [2] Object Management Group, *Model Driven Architecture –a Technical Perspective*, Document no. ormsc/01-07-01 July 2001.
- [3] P. Allen, S. Frost., *Component-Based Development for Enterprise Systems – Applying the SELECT Perspective*, Cambridge University Press/SIGS Cambridge, 1998.
- [4] D. D'Souza, A. C. Wills, *Objects, Component and Frameworks with UML, The Catalysis Approach*, Addison-Wesley, 1999.
- [5] J. Cheesman, J. Daniels, *UML Components: A Simple Process for Specifying Component-Based Software*, Addison-Wesley, 2001.
- [6] Object Management Group, *UML Profile for Enterprise Distributed Object Contributing (EDOC)*, Document no. ptc/02-02-05. 2002.
- [7] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Peach, J. Wüst, J. Zettel, *Component-Based Product Line Engineering with UML*, Addison-Wesley, 2001.
- [8] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
- [9] K. C. Wallnau, S. A. Hissam, R. C. Seacord, *Building Systems from Commercial Components*, Addison-Wesley, 2001.

Chapter 3

SCARLET: Light-Weight Component Selection in BANKSEC

Neil Maiden and Hyoseob Kim

Centre for HCI Design, City University London, UK

Abstract: This chapter reports the results of ongoing research into component-based software engineering (CBSE) in the European banking sector as part of the EU-funded BANKSEC project. The importance of complex non-functional requirements such as dependability and security presents new challenges for CBSE. The chapter describes SCARLET, an innovative process that enables the selection of components that satisfy such non-functional requirements. It presents 6 meta-requirements that we identified at the beginning of BANKSEC to inform SCARLET's design, and how SCARLET has been implemented to meet these requirements.

Key words: Requirements engineering, software component selection, use cases

1. REQUIREMENTS ENGINEERING FOR COMPONENT SELECTION

The European Union's Framework V programme is funding a series of new research initiatives in component-based software engineering (CBSE). BANKSEC is a 24-month research project that is investigating component-based dependable systems for the European banking sector. Dependable systems such as internet banking generate new research challenges for CBSE, in particular how to select components that, when implemented in an architecture, satisfy complex non-functional requirements such as security and reliability. This paper presents SCARLET (Selecting Components Against Requirements), a light-weight process that has been designed to satisfy 6 meta-requirements that we have identified for selecting components that satisfy such non-functional requirements. The resulting process utilises an

extended form of use cases to enable more integrated requirements specification and component evaluation.

Component-based software provides new opportunities for the efficient development of secure and dependable banking systems. However, new techniques are needed for specifying dependability requirements for components, for developing trusted application frameworks that can accommodate non-trusted components, and for designing and developing the necessary software and process infrastructure for component selection and integration. Indeed, the need to handle dependability requirements and application frameworks imposes new meta-requirements on the component selection process itself. We are delivering new processes that meet these requirements to assist European banks to build systems that meet these non-functional requirements using components. We are researching and implementing a process-driven software environment for component selection, and validating it in banking applications such as corporate lending support. To achieve this we have integrated previous research in PORE [1], a method for COTS (Commercial Off-The-Shelf) software package selection, with results from other requirements engineering research projects, to deliver innovative processes to European banks. The outcome of this research is SCARLET, an innovative requirements engineering process for components-based dependable systems that is being evaluated during trials with BANKSEC's banking partners.

This chapter presents 6 essential meta-requirements that have driven design of SCARLET. Section 2 describes SCARLET in the wider BANKSEC component procurement process. Section 3 presents the 6 meta-requirements that have informed the design of SCARLET and SCARLET's features that, we believe, meet these meta-requirements. Section 4 outlines the SCARLET process, then Section 5 describes how SCARLET utilises measurable fit criteria of stakeholder requirements to deliver more objective and streamlined component evaluation. Section 6 outlines the software prototypes being developed to support SCARLET, and the chapter ends with current trials of SCARLET in European banks.

2. SCARLET'S BASIC FEATURES

SCARLET is one of 3 key processes in BANKSEC's component-based systems development process shown in Figure 1. It assumes inputs and delivers outputs essential to the downstream component assembly process:

- In BANKSEC we determine high-level stakeholder and system requirements that are independent of component selection process using the C-Preview method [2]. We also use C-Preview to establish a high-level

system architecture that imposes technical requirements for component selection in SCARLET;

- In component assembly we design, implement and integrate the software system from the system architecture and selected software components. BANKSEC is developing an application framework specific to the banking sector to enable effective component integration.

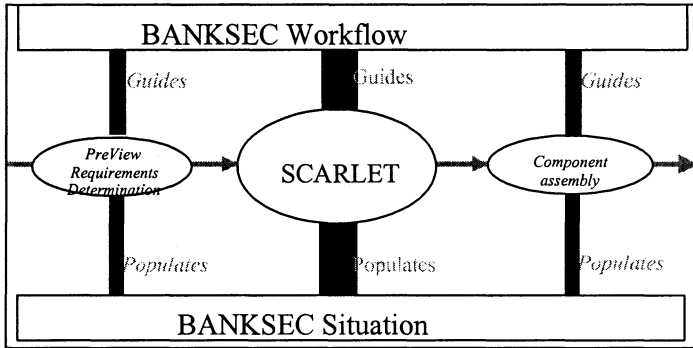


Figure 1. BANKSEC's component-based systems development process

2.1 Process Integration in BANKSEC

We define process integration in BANKSEC in terms of the inputs and outputs of the processes to ensure that process pre- and post-conditions are satisfied. BANKSEC's workflow engine, a workflow management system called FORO [3], supports the management of enterprise-wide processes and their constraints in a co-operative, distributed environment. FORO was originally developed and distributed by SchlumbergerSema. It supports the whole process from the analysis and development to the execution of workflow processes. Workflow analysis is carried out using a simple graphical notation supported by the design tool. Once the design is done, FORO provides a number of tools to implement the process and execute it. Information integration is defined in terms of the information that is manipulated by and exchanged between these processes, formalised in terms of BANKSEC's situation meta-model described in Maiden et al. [4].

2.2 SCARLET's Basic Processes

One of the most important and unique characteristics of SCARLET is that decision-making to select or reject software components drives the acquisition, modelling and validation of stakeholder requirements, so that decisions can be made in the right order at the right time. This means that, to

some degree, requirements determination is subordinated to component selection. We believe that this is a novel feature of SCARLET.

SCARLET prescribes 4 essential decision-making goals, which are to reject candidate components according to non-compliance with different types of customer requirements:

1. Simple customer requirements – high-level services and functions, requirements on the supplier or procurement contract, basic features of the software component such as price and source, and adherence to international standards;
2. Simple customer requirements that require access to the software component through demonstration or use – lower-level services and functions, and demonstrable attributes of the software component such as interface features;
3. Simple customer requirements that are demonstrable through short-term trial use of the software component – non-functional requirements such as performance measured as speed of response, throughput, usability and training;
4. More complex customer requirements with dependencies to other requirements and legacy systems – non-functional requirements that require more extensive trial use, such as maintenance and reliability requirements, and inter-dependencies with other software components, systems and legacy systems.

The rationale for this decision-making sequence is a simple and pragmatic one – to make the right and most simple decision at the most appropriate time using reliable information that is available to the selection team. As such the sequence relies on a set of assumptions – that information about components that enables the team to determine compliance with simple customer requirements is more readily available than the information needed to assess compliance with non-functional requirements. The sequence is similar to the decision-making process in the original PORE method, but it has been refined and improved in the light of the PORE trials [5] and research into decision-making methods.

SCARLET prescribes 4 processes to achieve these decision-making goals:

1. Acquire information about stakeholder requirements, software components, suppliers and procurement contracts;
2. Analyse acquired information for completeness and correctness;
3. Use this information to make decisions about component-requirement compliance;
4. Reject one or more candidate products as non-compliant with stakeholder requirements.

Figure 2 depicts these 4 processes graphically. The achievement of the processes is a broad sequence, in which the first process is acquisition of information from stakeholders and the last is selection of one or more

candidate components, but the sequence of the intervening processes is not predetermined and each process can be repeated many times.

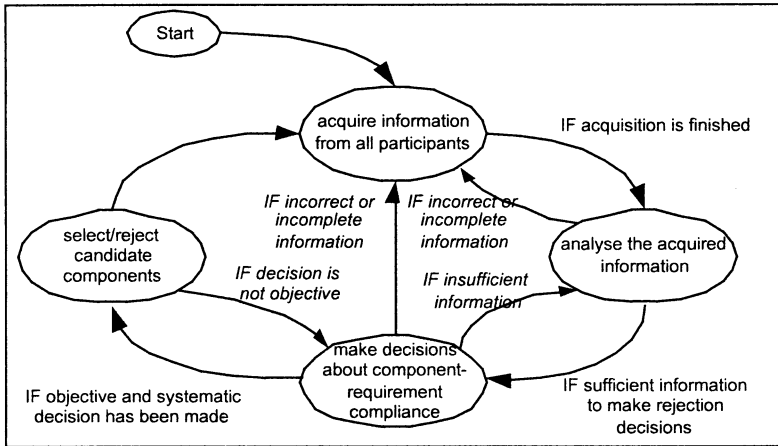


Figure 2. SCARLET's Situated-Dependent Decision-Making Process

However, developing new representation and techniques to implement this process in an effective and usable form is a research challenge. To address it, the BANKSEC consortium designed new representations and techniques to meet 6 key meta-requirements described in the next section.

3. A RIGOROUS AND LIGHT-WEIGHT COMPONENT SELECTION PROCESS

The integration of processes from disciplines as diverse as requirements engineering, component testing and multi-criteria decision-making can often lead to redundancies in their use as they have not been designed to work together. To inform the design of SCARLET, we have identified 6 meta-level requirements that SCARLET's representations and techniques need to satisfy:

1. Avoid duplicating artefacts and information used in different techniques, in order to keep the process lean, mean, and hence usable;
2. Provide one representation of stakeholder requirements and software components that can be used for different purposes including requirements acquisition, component evaluation and decision-making about component selection and rejection;
3. Design flexible processes that recognise the wide diversity and applicability of situations in which software components are selected;

4. Design processes that can be tailored to be successful in the different project environments in which software components are selected, and for the types of software components that are selected;
5. Enable a procurement team to make quick and objective decisions about each component's compliance with one or more requirements – establishing compliance is essential for effective decision-making;
6. Deliver flexible and tailored guidance for processes from diverse disciplines to a selection team in an effective manner.

We have designed SCARLET to satisfy these meta-requirements, and aim to demonstrate this satisfaction in BANKSEC trials. Table 1 presents the key features of SCARLET's design that enable it to meet each of the 6 meta-requirements.

Table 1. Key SCARLET features that satisfy its meta-requirements

Meta-requirements	SCARLET's Features
1. Avoid artefact and information duplication	A single representation to enable the acquisition and testing of requirements
2. Develop representations of requirements and components that can be used for different purposes with techniques from different disciplines	Extended use case specifications designed for effective requirements specification and component evaluation
3. Design flexible processes that can be used in diverse component selection processes	Situated process guidance that is underpinned with knowledge about when and how to use different techniques
4. Component selection processes that can be tailored to fit different procurement projects	A parameterised SCARLET process
5. Quick and effective decision-making about component's compliance with stakeholder requirements	The extension of VOLERE's measurable fit criteria to deliver stakeholder requirements that can be used to demonstrate component compliance with each requirement
6. Deliver flexible and tailored process guidance to procurement teams in a form that can be used	Integrated BANKSEC software environment linked to requirements and component information repositories that offers process guidance to selection teams

Each of SCARLET's 6 features are described in turn.

Avoid artefact and information duplication: Selecting software components that are compliant with stakeholder requirements is a complex process. It combines knowledge and skills from requirements engineering, software architectures, component evaluation, system testing and multi-criteria decision-making. However the integration of requirements, modelling, evaluation, testing and decision-making techniques is often cumbersome and difficult to achieve. New multi-purpose representations of requirements and components are needed to avoid artefact and information duplication. In SCARLET we have chosen use cases to integrate information about requirements, their fit criteria and component evaluations in an effective form. SCARLET's requirements, modelling, evaluation, testing and decision-

making techniques are all related to this one information artefact type, thus avoiding duplication of artefacts and information. The VOLERE shell [6], used to represent stakeholder requirements, is integrated into the use case artefact.

Develop a multi-purpose representation: A use case offers a unique structured representation for stakeholder requirements that affords both effective requirements specification and component evaluation. Use cases structure stakeholder requirements about the new system in a form that makes them amenable for test case generation during component evaluation. Use cases also define system boundaries that enable us to write context-sensitive stakeholder requirements, thus making them more testable. We can also evoke use cases as new stakeholder requirements are determined and added to each use case without the need to restructure the requirements, as it is the use case that provides the information structure for the requirements.

Flexible component selection processes: There are a large number of situations that may arise at any point in a selection process, and many techniques from different disciplines are available to achieve each situation. Therefore, the order in which a selection team acquires stakeholder requirements and information about component features or makes decisions about component selection and rejection is situation-driven, that is it depends on the current state of the stakeholder requirements, selected components and evaluation process. In SCARLET we define a situation as a graph of contexts that represent decisions to pursue a specific goal in a specific situation. A situated process is a process that pursues a specific component selection goal in a defined situation using techniques that are applicable in and tailored to that situation.

Tailored procurement processes: Component selection must also be customisable to handle the resources and time available for component selection. Component selections can vary widely according to the size and nature of the software 'component', from an ERP package to a JavaBean component, the number of candidate components, from 2 to over 100, and the time to make a decision, from days to years. A one-size-fits-all selection process will not work in most procurements. Therefore, SCARLET is parameterised to allow a component selection team to remove one or selection iterations, and to select techniques for situations according to these parameters.

Quick and effective decision-making: The ability to make decisions about each component's compliance to stakeholder requirements is critical for effective component selection. As a typical process will involve a large number of components and 100s of stakeholder requirements, teams need quick, clean and effective techniques to determine component-requirement compliance. In SCARLET we believe that preparation is essential, and recommend the development of precise and testable requirements before

evaluation begins, to make the process as objective as possible. To achieve this we build on tried-and-tested techniques requirements engineering techniques, and in particular measurable fit criteria from VOLERE [6] that have been tailored for evaluating software components. Instead of using quantitative measures to record the degree to which a component satisfies a requirement, as in the Weighted Average Sum or OTSO [7] methods, SCARLET guides a team to record each component as either compliant or not with each requirement. Compliance evaluation in SCARLET is described at length in Section 5 of this chapter.

Delivering flexible and tailored process guidance: Selecting software components is complex. The differences in the length and nature of, and resources available for different selection processes introduce variety into the process. Our use of situated process guidance to handle this variety demands complex new mechanisms for delivering the process guidance. In SCARLET we propose an integrated BANKSEC software environment linked to requirements and component information repositories. The key feature of this environment is the integration of process guidance to product models through an innovative concept called a process chunk. Our definition of a chunk is based on the process view of the ESPRIT III 6353 'NATURE' process modelling formalism [8], [9]. As we have defined, a process is described as a graph of contexts. A context represents the decision to pursue a specific goal in a specific situation. A situation is a condition over the state of the model being manipulated by the process. A process chunk is process guidance to be implemented in a context, which is to meet a goal in a specific situation. SCARLET combines such chunks together in different sequences, using its FORO workflow engine, to form different processes to achieve the 4 different decision-making goals.

The remainder of this chapter describes the SCARLET process in more detail, and demonstrates how the process satisfies the 6 meta-requirements.

4. PUTTING IT ALL TOGETHER IN SCARLET

SCARLET's process structure and use of artefacts is depicted in Figure 3. The process is divided into 4 key stages that enable different decisions to be made about component selection and rejection. At each stage SCARLET provides specific advice in terms of:

- The types of requirement that enable decision-making at this stage;
- The most effective techniques to determine these requirement types;
- The essential use cases for specifying requirements of these types;
- The measurable fit criteria to enable effective component evaluation against these requirements.

As the process advances the selection team acquires different types of requirements with different techniques, integrates them into the evolving use cases in different ways, and develops different types of fit criteria to enable effective component evaluation. The process encourages the team to acquire the requirements in the order prescribed to facilitate effective decision-making. In this section the first 3 concept types are described.

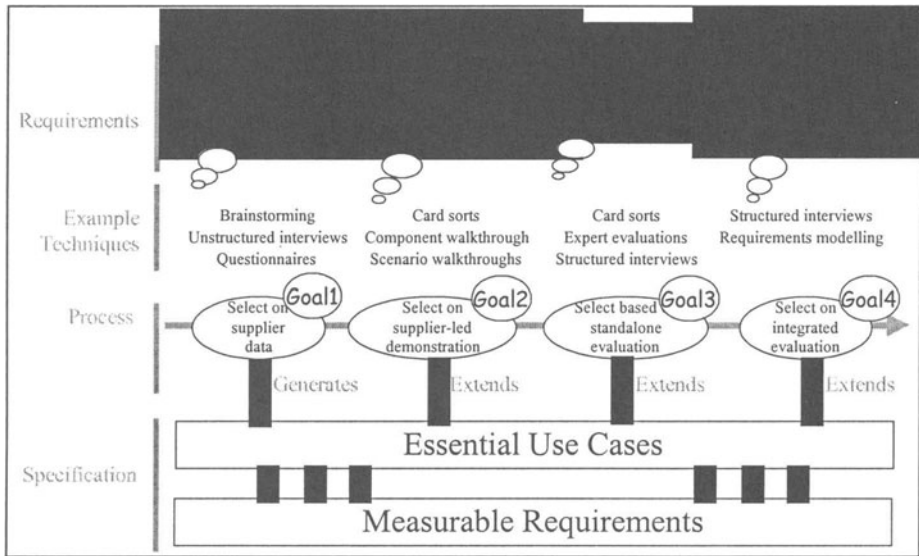


Figure 3. An overview of SCARLET

Requirements types: The team acquires different types of requirements to enable the team to make decisions in a simple and pragmatic sequence – that is to make the right and most simple decision at the most appropriate time in the procurement process using reliable information that is available to the team. SCARLET uses a type model derived from existing taxonomies of requirements, and in particular the VOLERE approach [6] to guide requirements acquisition. At the beginning of the SCARLET process, the team will acquire supplier, contract, legal, portability, operational, capacity and simple functional and information-type requirements to make simple decisions to reject large numbers of non-compliant software components. Functional and information requirements that are more difficult to test without access to the components are acquired and tested later in the process. In contrast reliability, availability, maintainability, security and safety-type requirements are tested for component compliance at the end of the process during longer component trials. More details of this requirements taxonomy and the process guidance are found in [10].

Acquisition techniques: Requirements engineers face a major problem when choosing techniques for acquiring stakeholder requirements. The problem is not that there is a lack of techniques, since a wide range exists from the ethnographic to the constructivist. Rather, little guidance is available to choose techniques, to plan a systematic, well-grounded acquisition programme, or even to sequence techniques. Indeed, many requirements engineers are unaware of the techniques that are available. ACRE – ACquiring REquirements [11] – recognises that different requirements acquisition techniques have different strengths and weaknesses. SCARLET implements ACRE’s process guidance within its own selection process, and provide specific techniques to acquire specific types of requirements at different stages of the process.

Essential use cases: Use case specifications evolve to support decision-making at different stages in the process. Requirements discovered to enable the first round of component rejection leads to the development of a use case specification that is then evolved throughout the remainder of the process. We adopt Constantine and Lockwood’s [12] notion of essential use cases to model the required behaviour of the future system, but extend it to integrate requirements statements and use cases within a single and integrated artefact. Furthermore, the use of use cases in the selection of software components avoids one of the most well-documented weaknesses of use cases, that is making premature design decisions about the future system. In BANKSEC the decision to implement one or more software components reflects high-level decisions to automate functions and services, and the consequences of these decisions can be written into the use cases to the inform of automated actions. As such, use cases offer an ideal but under-exploited solution to requirements specification for component selection.

5. QUICK AND EFFECTIVE COMPONENT EVALUATION

Effectively evaluating the degree of compliance of components to requirements is at the heart of the SCARLET process. Evaluation can appear trivial but is fraught with difficulties. Teams often use weighted scoring methods such as the WAS to score degrees of compliance with each requirement. This is intuitively appealing but problematic. Offering a selection of range of compliance values creates numerous opportunities for disagreement between team members. Furthermore, in one such selection process, the post-evaluation analysis of the resolution of such disagreements revealed a trend towards agreement with the scores of one consortium member [1]. SCARLET seeks to reduce the risk of such disagreements and bias during evaluation. Moreover, scoring methods such as the WAS

(Weighted Average Sum) do not fit well with the concept of measurable fit criterion from requirements engineering. Use of logical compliance tests from requirements engineering introduces more rigour into component evaluation and can reduce the scope for subjective judgement during component evaluation.

To avoid these problems in SCARLET we adopt VOLERE's measurable fit criterion concept [6] to determine precise and testable requirements to drive objective component evaluation. A fit criterion defines the minimum acceptance criterion, which is the minimum level of satisfaction of acceptance of the requirement, expressed as a quantifiable or logical test. Dependability requirements common to banking applications must often be decomposed to deliver these quantifiable and logical tests. Furthermore, in SCARLET, the selection team must take into account the relevant evaluation strategies used to test for requirements compliance and tailor the fit criteria accordingly. Using this approach a team evaluates each component as either compliant or non-compliant with each requirement according to evidence available from the evaluation.

SCARLET's innovative component evaluation approach is depicted in Figure 5. It combines results from direct requirement-component evaluations that give independent compliance scores with results of comparative analyses of candidate components using the Analytic Hierarchy Process, or AHP for short [13]. A typical evaluation would proceed in the 3 stages shown in Figure 4. Firstly, we use the AHP to prioritise requirements – essential when selecting between components – using a technique that recognises the difficulties of comparing criteria of different types. Rankings are obtained through paired comparisons of requirement that are then converted to normalised rankings using the eigenvalue method, which means that the relative rankings of alternatives are presented in ratio scale values which total one [13]. The AHP then enables the vertical computation of horizontal comparison ratios. The resulting requirements rankings are both transitive and complete.

Secondly SCARLET uses these ranked requirements to test for component compliance using measurable fit criteria. This evaluation against individual requirements using well-defined tests and units such as time or mean-time between failures gives independent ratings of compliance. From the set of requirement-component compliance we can determine the subset of components satisfy the ranked requirements, but the relative degrees of fit is not known. This is where the AHP is used again.

SCARLET then applies the AHP to undertake pair-wise comparisons of the subset of compliant components with a subset of the requirements to provide a ranking of the compliant components. The AHP adds rigour to the evaluation by detecting inconsistencies between pair-wise comparisons with eigenvalue method. The resulting rankings inform selection and rejection

decision-making in SCARLET, although the selection team needs to be aware that the quantification of subjective, pair-wise comparisons may imply a level of precision not available with the AHP. Of course this cannot always be done effectively for large numbers of stakeholder requirements, so SCARLET encourages the team to focus on requirements that enable effective component discrimination, as described in [1].

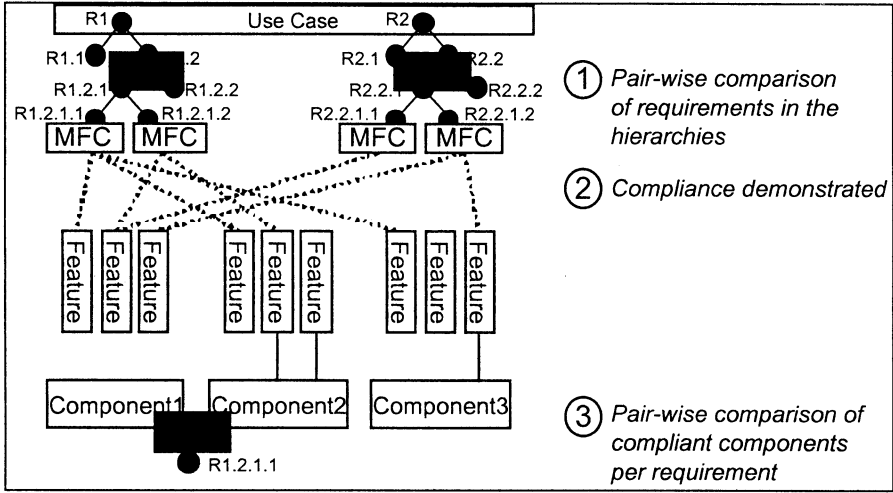


Figure 4. A combined compliance and pair-wise approach to component evaluation

We believe that this combined decision-making approach will deliver quicker and more effective component evaluation, thus meeting SCARLET's 5th meta-requirement. Its strength is its belt-and-braces approach that combines independent objective assessments of compliance with different requirements types using measurable fit criteria with pair-wise comparisons of requirements and of component compliance with these requirements to provide a ranking of candidate components.

6. SCARLET AND BANKSEC SOFTWARE TOOLS

We are building a software prototype to implement the 6th SCARLET meta-requirement using workflow, data base and process guidance technologies in the BANKSEC environment architecture shown in Figure 5.

BANKSEC's workflow engine is based on 3 models of a workflow. The process model defines what it is necessary to do. The information model defines the information to use in the process. The organisation model defines who must perform the process. The process model allows us to define synchronised actions between tasks, like operations with process variables,

start a case, or any of the FORO API available actions. We use FORO tools to model, apply and test the workflow. The process designer tool allows us to design the SCARLET process in graphical form. Furthermore, FORO can be integrated with other requirements engineering or decision-making tools using a standard CORBA-IDL interface. Its API allows the selection team to build ad-hoc tools or integrate off-the-shelf ones to support the wider selection process.

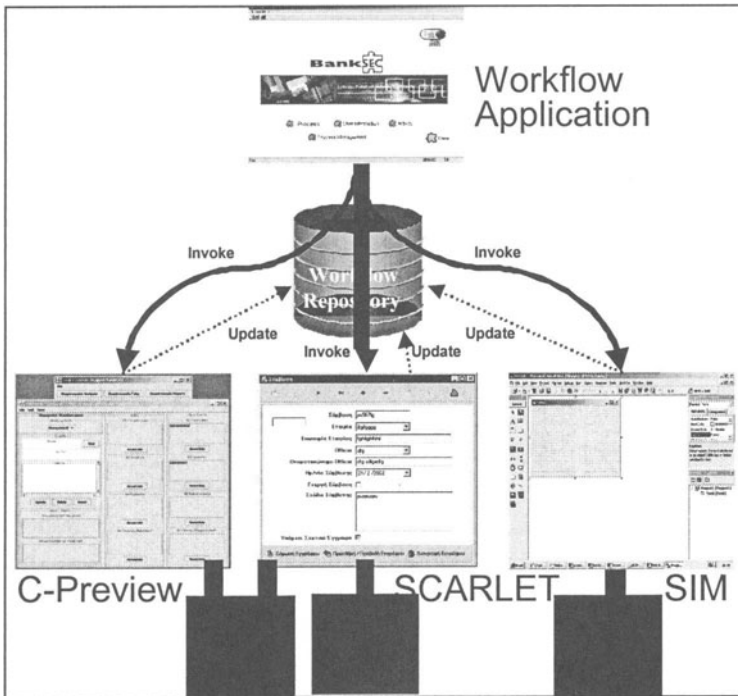


Figure 5. Overview of the BANKSEC environment architecture

FORO's workflow application invokes 3 BANKSEC tools shown in Figure 5. It invokes the C-Preview requirements tool for modelling high-level system requirements and architectures that are input to the SCARLET process. It also invokes the systems integration manager (SIM) that use BANKSEC's application framework to support component assembly and integration using components selected with SCARLET. The workflow application also invokes SCARLET's process advisor tool to provide goal-driven process guidance using the contents of the workflow repository and local process guidance informed using the current state of the requirements and components data bases [14].

The BANKSEC workflow engine and software tools are currently under development, and will soon be available to software developers once user

trials have been completed. We look forward to readers contacting us with expressions of interest in SCARLET's finished processes and software tools.

7. CONCLUSIONS AND TRIALS

This chapter provides an overview of BANKSEC's process for component selection and the rationale for its design in terms of its 6 meta-requirements. We are currently evaluating SCARLET by transferring it to 2 European banks, Banca Popolare di Sondrio in Italy and Eurobank in Greece. The banks are using SCARLET to guide the selection of software components for secure and dependable banking applications such as home internet banking and corporate loan management. In preparation for these trials, key stakeholders from both banks have had input into the requirements and design for the SCARLET process, to increase the likelihood of its successful take up. Each trial will last 6 months, and will deliver SCARLET in different forms to selection teams in the 2 banks. We look forward to reporting on the results of these trials, and in particular the effectiveness of the implemented FORO workflow engine in the near future.

ACKNOWLEDGEMENTS

The authors wish to thank all of the BANKSEC partners for their inputs to and support for the work reported in this chapter. The work is funded by the EU-funded IST-1999-20711 BANKSEC project.

REFERENCES

- [1] Maiden N.A.M. and Ncube C., 1998, "Acquiring Requirements for Commercial Off-The-Shelf Package Selection", *IEEE Software*, 15(2), 46-56.
- [2] Kotonya G. and Sommerville I., 1998, "Requirements Engineering Processes and Techniques", John Wiley & Sons.
- [3] Gutiérrez G.S., 1999, "The WIDE Project: Final Report", ESPRIT Project 20280, May, 1999.
- [4] Maiden N.A.M., Kim H. and Ncube C., 2002, "Rethinking Process Guidance for Software Component Selection", *Proceedings 1st International Conference on COTS-Based Software Systems, Lecture Notes on Computer Science LNCS 2255*, Springer-Verlag, 151-164.
- [5] Ncube C. and Maiden N.A.M., 2001, "Selecting the Right COTS Software: Why Requirements are Important", In *Component-Based Software Engineering: Putting the Pieces Together* (eds. George T. Heineman and William T. Councill, Addison-Wesley, 467-478.
- [6] Robertson S. and Robertson J., 1999, "Mastering the Requirements Process", Addison-Wesley-Longman.

- [7] Konito, J., 1996, "A Case Study in Applying a Systematic Method for COTS Selection", Proceedings 18th International Conference of Software Engineering, IEEE, Computer Society Press, 201-209.
- [8] Rolland C. and Grosz G., 1994, "A General Framework for Describing the Requirements Engineering Process", IEEE Conference on Systems, Man and Cybernetics, CSMC94, IEEE Computer Society Press.
- [9] Plihon V. and Rolland C., 1995, "Modelling Ways of Working". Proceedings 7th International Conference on Advanced Information Systems Engineering, CAiSE'95, Springer Verlag.
- [10] Maiden N.A.M. and Kim H., 2001, "SCARLET: Process Advice for Component-Based Software Engineering", Technical Report, Centre for HCI Design, City University London, November 2001.
- [11] Maiden N.A.M. and Rugg G., 1996, "ACRE: Selecting Methods For Requirements Acquisition", Software Engineering Journal, 11(3), 183-192.
- [12] Constantine L.L. and Lockwood L.A.D., 1999, "Software for Use", Addison-Wesley-Longman.
- [13] Saaty T.L., 1990, "The Analytic Hierarchy Process", New York: McGraw-Hill.
- [14] Maiden N.A.M. and Kim H., 2002, "SCARLET: Providing the Right Advice at the Right Time when Selecting Software Components", Technical Report, Centre for HCI Design, City University London, March 2002.

Chapter 4

Built-in Contract Testing for Component-Based Development

Hans-Gerhard Groß¹, Colin Atkinson¹, Franck Barbier², Nicolas Belloir² and Jean-Michel Bruel²

¹IESE, Germany; ²LIUPPA, University of Pau, France

Abstract: Assembling new software systems from prefabricated components is an attractive alternative to traditional software development practices. However, the expected reductions in development time and effort will only arise if separately developed components can be made to work effectively together with minimal effort. Lengthy and costly in-situ verification and acceptance testing directly undermines the benefits of heterogeneous components and late system integration. This chapter describes an approach that reduces manual system verification effort by equipping components with the ability to check their execution environment at run-time. When deployed in a new system, built-in contract test components check the contract-compliance of their server components, including the run-time system, and thus automatically verify their ability to fulfil their own obligations. The chapter first considers the principles behind built-in contract testing, and then describes how built-in testing can be made a natural part of component-based development.

Key words: In-situ deployment test, testing interface, tester component, test modeling

1. INTRODUCTION

The vision of component-based development is to allow software vendors to avoid the overhead of traditional development methods by assembling new applications from high-quality, prefabricated, reusable parts. Since large parts of an application may therefore be constructed from prefabricated components, it is expected that the overall time and costs involved in application

development will be reduced, and the quality of the resulting applications will be improved. This expectation is based on the implicit assumption that the effort involved in integrating components at deployment time is lower than the effort involved in developing and validating applications through traditional techniques. However, this does not take into account the fact that when an otherwise fault-free component is integrated into a system of components, it may fail to function as expected. This is because the other components to which it has been connected were intended for a different purpose, have a different usage profile in mind, or are themselves faulty.

Current component technologies can help to verify the syntactic compatibility of interconnected components (i.e. that they use and provide the right signatures), but they do little to ensure that applications function correctly when they are assembled from independently developed components. In other words, they do nothing to check the semantic compatibility of interconnected components, so that the individual parts are assembled into meaningful configurations. Software developers may therefore be forced to perform more integration and acceptance testing in order to attain the same level of confidence in the system's reliability. In short, although traditional development time verification and validation techniques can help assure the quality of individual components, they can do little to assure the quality of applications that are assembled from them at deployment time.

The correct functioning of a system of components at run time is contingent on the correct interaction of individual pairs of components according to the client/server model. Component-based development can be viewed as an extension of the object paradigm in which, following Meyer [1], the set of rules governing the interaction of a pair of objects (and thus components) is typically referred to as a contract. This characterizes the relationship between a component and its clients as a formal agreement, expressing each party's rights and obligations. Testing the correct functioning of individual client/server interactions against the specified contract therefore goes along way towards verifying that a system of components as a whole will behave correctly. The approach described in this chapter is therefore based on the notion of building contract tests into components so that they can validate that the servers to which they are "plugged" at deployment time will fulfil their contract. Although built-in contract testing is primarily intended for validation activities at deployment and run-time, the approach also has important implications on the development phases of the software life-cycle. Consideration of built-in test artifacts needs to begin early in the design phase as soon as the overall architecture of a system is developed and/or the interfaces of components are specified. Built-in contract testing therefore needs to be integrated within the overall software development methodology. In this chapter we explain the basic principles behind built-in contract testing, and show how it can be integrated with, and made to complement, the KobrA method [2] intro-

duced in chapter two of this book as a model-driven approach to component based development. Here, we use the notion of a component in the same sense as it is outlined in this chapter. This is also discussed in more detail in [2].

The next section introduces the terminology and artifacts upon which built-in contract testing is based. The main artifacts are tester components and testing interfaces which represent the respective sides of a client/server relationship between two components. The following sections then describe how this technology can extend the KobrA development method. Section 4 discusses a few practical implications that must be considered when applying the technique in real software projects. Finally, Section 5 sums up and concludes the chapter.

2. BUILT-IN TESTING ARTIFACTS

Meyer [1] defines the relationship between an object and its clients as a formal agreement or a contract, expressing each party's rights and obligations in the relationship. This means that individual components define their side of the contract as either offering a service (this is the *server* in a client-server relationship) or requiring a service (this is the *client* in a client-server relationship). Built-in contract testing focuses on verifying these pairwise client/server interactions between two components when an application is assembled. This is typically performed at deployment time when the application is configured for the first time, or later during the execution of the system when a reconfiguration is performed.

2.1 Built-in Tester Components

Configuration involves the creation of individual pairwise client/server relations between the components in a system. This is usually done by an outside "third party", which we refer to as the context of the components. This creates the instances of the client and the server, and passes the reference of the server to the client (i.e. thereby establishing the clientship connection between them). This act of configuring clients and servers is represented through the KobrA style «acquires» stereotype illustrated in Figure 1. The context that establishes this connection may be the container in a contemporary component technology, or may simply be the parent object.

In order to fulfil its obligations towards its own clients, a component that acquires a new server must verify the server's semantic compliance to its clientship contract. It means the client must check that the server provides the service that the client has been developed to expect. The client is therefore augmented with in-built test software in form of a tester component as shown in Figure 1. This is called a *server tester component*, and is executed when the

client is configured to use the server [3]. In order to achieve this, the client will pass the server's reference to its own server tester component. This is represented through an «acquires» association between the server tester component and the server in Figure 1. If the test fails, the tester component may raise a contract testing exception and point the application programmer to the location of failure.

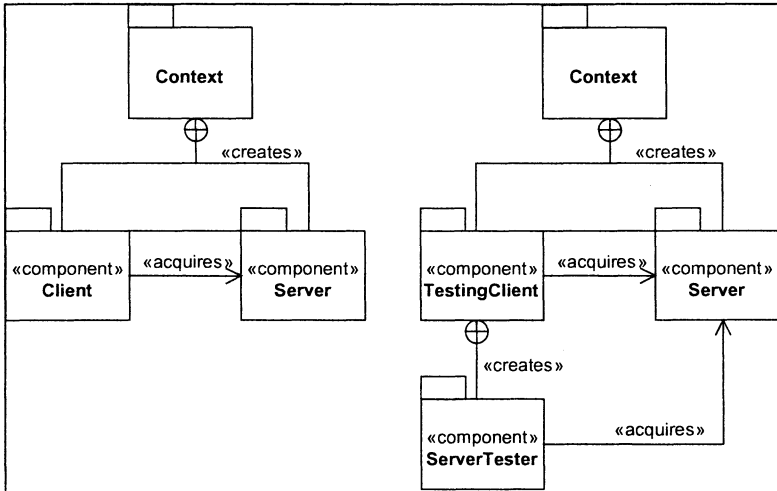


Figure 1. KobraA style component containment hierarchy without and with built-in contract testing

A test involves the invocation of the methods of an associated component with predefined input values and the checking of the returned results against the expected results. The input data and expected results are referred to as a test case. Under the object paradigm, a test case also often not only involves the checking of the results of the method invocations but also the checking of the correctness of the state transitions according to the external states. A test suite for a server tester component therefore contains a number of test cases that are developed according to distinct testing criteria, for example the coverage of the state transition model or the coverage of the functional specification. These are typically augmented with tests according to equivalence-class partitioning and boundary value analysis [2], [4], [5].

We call a client that owns a tester component and performs a contract test on its acquired server a *testing client* or a *testing component* [3].

2.2 Build-in Testing Interfaces

The object-oriented and as a consequence the component-based development paradigm builds on the principles of abstract data types which advocate

to the combination of data and functionality in a single entity. State transition testing is therefore an essential part of component verification. In order to check whether a component's operations are working correctly it is not sufficient simply to compare their returned values with the expected values. The compliance of the component's externally visible states and transitions to the expected states and transitions according to the specification state model must also be checked. These externally visible states are part of a component's contract that a user of the component must know in order to use it properly. However, because these externally visible states of a component are embodied in its internal state attributes, there is a fundamental dilemma.

The basic principles of encapsulation and information hiding dictate that external clients of a component should not see the internal implementation and internal state information. The external test software of a component therefore cannot get or set any internal state information. The user of a correct component simply assumes that a distinct operation invocation will result in a distinct externally visible state of the component. However, the component does not usually make this state information visible in any way. This means that expected state transitions as defined in the specification state model cannot normally be tested properly.

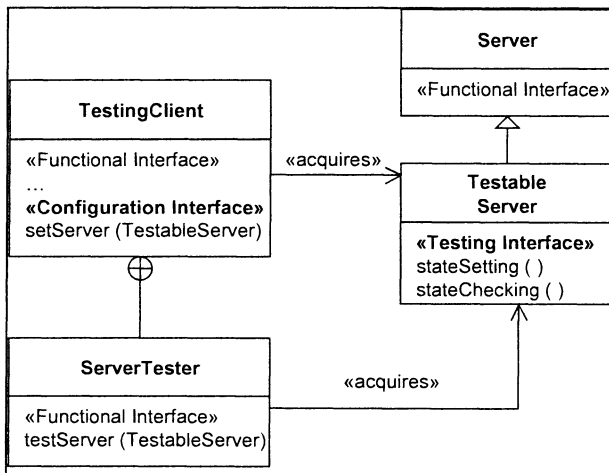


Figure 2. A testable server within the Kobra component hierarchy, and the organization of the testing client and the testable server

The contract testing paradigm is therefore based on the principle that components should expose their logical or externally visible (as opposed to internal) states by extending the normal functional server as displayed in Figure 2. A testing interface provides additional operations that read from and write to internal state attributes that collectively determine the logical states. These

auxiliary interface operations are usually derived through typical assertion checking techniques [5], [6], although for contract testing they are much more formally defined and applied, since they essentially become part of a component's normal functionality.

A testing interface augments the functionality of the tested server with state checking and setting operations. The state checking operations verify whether the component is currently residing in a distinct logical state (for verifying the post conditions of a test case). The state setting operations set the component's internal attributes to represent a distinct logical state (for satisfying the preconditions of a test case). State checking operations are more fundamental than state setting operations. The latter may often involve quite considerable development effort. Thus, in most cases state setting will be achieved by invoking the operations of the normal functional interface. Subsequently, the state checking methods may be used to verify that the preconditions (initial state) for a test case are satisfied.

Within the server tester component a test case may be applied in two alternative ways. In the first way the state setting operations, if applicable, are invoked to ensure the preconditions required for a test case, the tested operation is invoked with the predetermined input parameters according to the testing criterion, and finally, the state checking operations are invoked to verify the post conditions required for a test case. In the second way is applied when no state setting operations are provided by the tested component. Then, the operations of the component's normal functional interface have to be invoked to bring the component into the desired initial state for a test. Since these operations are part of the software that should be tested, the state checking operations have then to be invoked to verify the correct precondition for the application of a test case. Finally, the test method is called, and the state checking operations are used to verify the post conditions against the expected outcome.

We call a component that provides a testing interface and that is tested by a contract tester component a *testable server* or a *testable component* [3]. For example the *bank* component in Figure 3, becomes a *testing bank* since it performs a contract test on its associated server, the *converter* component.

2.3 Built-in Test Components

The distinction between clients and servers is only intended to refer to the roles that can be played in a pairwise interaction between two components. When viewed from a global perspective, individual components can, and usually do, play the role of both clients and servers. Any of the client/server relationships of components may be subject to contract tests. In the server role, a component provides a testing interface that supports the tests performed by its

client's tester components, and in the client role, the component owns tester components that use the testing interfaces of its associated servers.

We call a component that plays *both* roles (i.e. provides a testing interface to its clients *and* contains its own tester components to test its servers) a *Built-in Test (or BIT) component*. For example, the *converter* component in Figure 3 is a BIT component because it provides a testing interface to its client, the *testing bank*, and it owns a tester component that verifies the correctness of its server, the *testable lookup table*.

3. BUILT-IN CONTRACT TESTING WITH THE KOBRA METHOD

The previous section illustrated the artifacts that must be developed in order to apply built-in contract testing. This section describes how the built-in contract testing artifacts are realized, and how built-in contract testing supplements the Kobra development method that was introduced in chapter 2. Here, we focus on the testing methodology that is based on the Kobra development process. We extend the existing simple international banking (SIB) system that was used in chapter 2 to demonstrate how the method can be applied for test modeling, and for deriving contract testing components and interfaces. Chapter 2 also contains the Kobra specification for this example.

3.1 Built-in Testing Artifacts

The initial step is the identification of the pairwise client/server interactions that must be tested. These are the identified *acquires associations* between the components in the Kobra containment hierarchy.

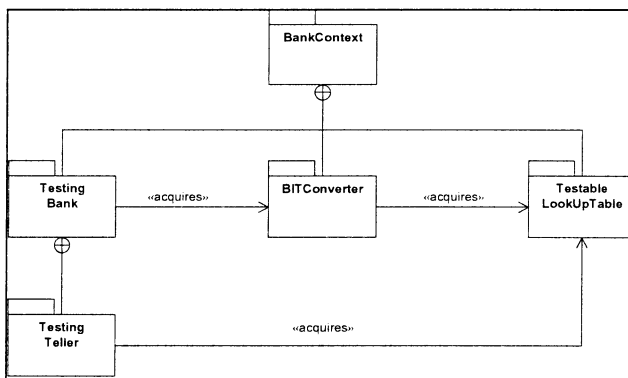


Figure 3. The architecture of the SIB system with contract testing artifacts represented by a Kobra containment tree

The model in Figure 3 represents the same version of the SIB system as that described in chapter 2 (i.e. version 6), but with contract testing components replacing the standard components. Components that are the source of an *acquires association* will contain server tester components (testing component; *testingBank*, and *testingTeller*, Figure 3). Components that are the target of an *acquires association* will provide a testing interface (testable component; *testableLookupTable* in Figure 3). Components that are the source and the target of *acquires associations* will do both, provide a contract testing interface and contain their own server testers (BIT component; *BITConverter*, Figure 3).

Each component that participates in an *acquires* relationship must be augmented with additional components or interfaces. In this instance we have extended the original functionality of the SIB components from chapter 2 with state setup and verification interfaces. Each testing component that extends the original functional component owns one or more server testers, and these are also associated with the tested server component. Essentially, this amounts to passing the reference of the server to the server tester. This is represented through the *acquires* relationships that originate at the components and the testers in Figure 4. They define the individual contract testing artifacts that must be developed for each component in the containment tree, and this illustrates how they are related to the original functional components. This is the way in which the KobrA method supports the parallel design and development of the test software together with the functional software. Every component may be associated with its respective testable and testing components according to which form of testing is required.

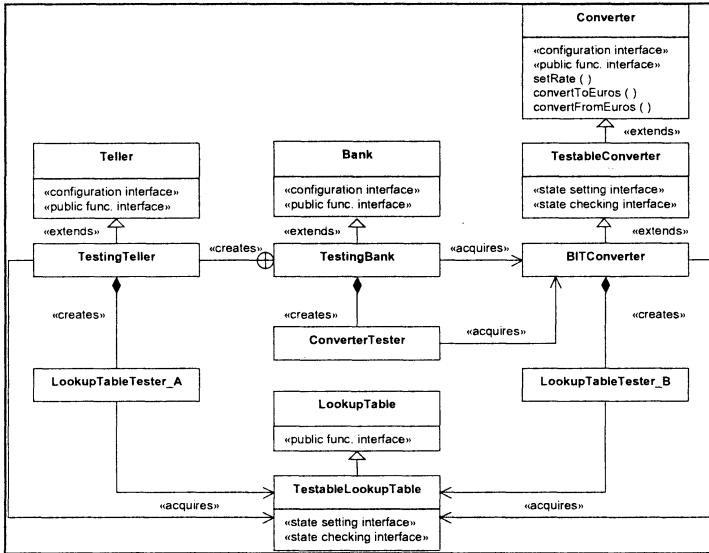


Figure 4. The class diagram for the SIB system with contract testing artifacts (structural model)

3.2 Contract Testing Development Process

In the previous section we determined which components of an application will be augmented with which contract testing artifacts. These are identified in Kobra's hierarchical containment tree and specified in the structural model. The resulting models comprise the extensions for testing as displayed in Figure 4. These two views, containment tree and structural model, are complementary and refine each other. In the following we will focus on how the contract testing artifacts are derived from Kobra's behavioral and functional specification models.

3.2.1 Development of the Testing Interface

The testing interface is developed together with a server component. It extends the testability of the server and it is primarily concerned with verifying and setting the internal state attributes according to the logical states. The logical states are part of a component's specification and are essential for using the component correctly. These states define the externally visible behavior of the component when distinct events occur. In Kobra they are represented through the specification behavioral model. This comprises a number of UML statechart diagrams, or state tables like the one depicted in Table 1 in the appendix. This defines the behavioral model of the *Converter* component. It comprises the events that map to transitions (*setRate*, *convertToEuros* and

convertFromEuros) plus the respective pre and post conditions according to the initial and final states of the state model. Each entry in the state table represents an “item of behavior” that must be verified when the component is deployed. This means that each of these entries represents the specification of at least one test case in a state transition tester component. We explain this in the following subsection. What is important here is that the state table represents the specification for the component’s testing interface.

Each item in the table is defined according to initial and final states plus the defined pre and post conditions that must be satisfied. In order to check whether the state model is implemented correctly, all these items must be executable in the form of tests. This means the component must be brought into a state that satisfies the expected preconditions, and the component’s post conditions must be verifiable. The testing interface will therefore comprise operations that make this feasible, and these are specified according to the requirements in the state transition table (pre and post conditions, plus initial and final states).

Table 2 (appendix) shows the operation specifications that verify the post conditions of the *Converter* and set its preconditions. These make up the testing interface. The special operation *ruOk()* (“are you o.k.”) transparently executes all internal assertion checking mechanisms that the component implements. This is determined through additional specification that is not explicitly represented in the behavioral model. For our *Converter* component this operation can be made to check that each entry in the converter is only stored once, for instance. Figure 5 displays the class diagram that represents this set up.

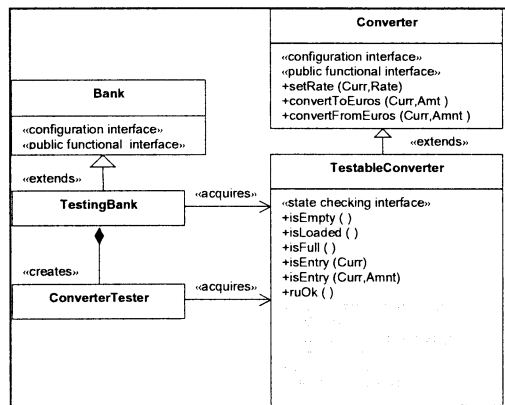


Figure 5. The class diagram for the SIB system with contract testing artifacts

The state setting operations may be designed in the same way as the state verification operations. However, their development can often involve con-

siderable effort that basically amounts to re-implementing the full functionality of the component. This is typical for abstract data type containers that store and manage data. In this particular example we would therefore recommend to use the normal functional interface in order to set the preconditions, and then use the state verification operations to check that the preconditions have been set correctly. How the testing interface is used by a tester component is the subject of the next section.

3.2.2 Development of the Server Tester Component

The server tester component is developed according to the needs of the client which it will be built into. This server tester checks what the client has been developed to expect from its server, including functionality as well as behavior. The Kobra method provides a specification for each of the associated server components that appear in the client's realization. According to the principles of polymorphism, it is not a particular component that the client expects, but any component that conforms to the client's specification (i.e. its contract). The client will therefore accept a new server not only on the basis that it provides the right syntactic signatures, but more importantly that it implements the contract in a semantically correct way.

Kobra's behavioral specification contains everything that is required to set up a test suite that focuses on state transition testing, and Kobra's operation specification and algebraic specification contain all the information that is necessary to set up the functional testing. Functional testing and state transition testing are related because an operation invocation always also triggers a state transition (at least one that is reflexive). The *ConverterTester* component that is built into the *TestingBank* will therefore embrace a number of test cases that are based on the *Converter*'s behavioral and functional specifications. The *Converter*'s behavioral model is represented through the state transitions in Table 1 (see appendix).

Each of the fifteen items in Table 1 maps to a single test case for a *ConverterTester* component with the initial state and precondition, the event that triggers a transition, and the expected final state with the expected post conditions. Moreover, it represents a minimal but full set that covers the functional specification (i.e. every defined element of functionality of the component is tested) as well as the behavioral specification (i.e. every defined state transition of the state model is covered). For example, the test case #4 in Table 1 will be designed in the following way:

```

initial state
loaded with n=2      setRate (Curr1, Rate1)
                    setRate (Curr2, Rate2)
                    isLoaded ( ) : true

```

```

isEntry (Curr1, Rate1) : true
isEntry (Curr2, Rate2) : true
or alternatively with the state setting operation
loaded with n=2      setLoaded (2, {(Curr1,Rate1),(Curr2,Rate2)})
event, the precondition is that the parameter Curr2 is in the Converter
convertFromEuros    convertFromEuros (Curr2, Amount) : result

evaluation of final state and post conditions
loaded              isLoaded ( ) : true
return conversion   result equals expectedResult : true

```

The state model together with the functional model essentially defines a minimal test suite for checking the server. Additionally, any other test cases according to any arbitrary test criteria such as equivalence-class partitioning or boundary value analysis are conceivable. All these additional tests may be designed in the same fundamental way and use the same testing interface operations in combination with the normal functional interface operations of the server.

4. CONTRACT TESTING IN PRACTICE

So far, we have only considered the technical principles behind built-in contract testing and how the technology augments a mainstream model-driven development method. In this section we discuss the implications of the technology that must be considered when it is applied in real software projects.

4.1 Test Weight Selection

Since heavy tests are usually extensions of lighter tests, heavyweight tester components will usually contain lighter weighted tester components. In other words, the test method for a heavyweight tester will include a call to the next lightest component as well as the additional test cases that make it heavier. By hierarchically organizing test methods and tester components in this way, for example through an «extends» mechanism, the replication of test cases at runtime can be avoided. A configuration interface can be used to select the required test weight. In the same way that the configuration interface of the client sets up the connection to a server component, the tester component may be acquired according to the required type and thoroughness of testing. This means that testing components must provide a configuration interface that comprises set up operations for functional components as well as set up op-

erations for testing components. The class diagram that represents this situation for the *TestingBank* is depicted in Figure 6.

The size and thoroughness of test that will be built into a component at deployment time is dependent upon criteria such as:

- Time of the test: that is how often is a test performed and at what time(s) over the lifecycle of the application will the test be used, for example development time regression test, deployment-time configuration test, operation-time reconfiguration test.
- Origin of the component: in other words, did we get the component from a trustworthy source, for example in-group, in-house, or third party development?
- Mission criticality: for example is the application safety critical or not?
- Availability of resources: in other words, are we dealing with an embedded system with low memory, or an Internet application?

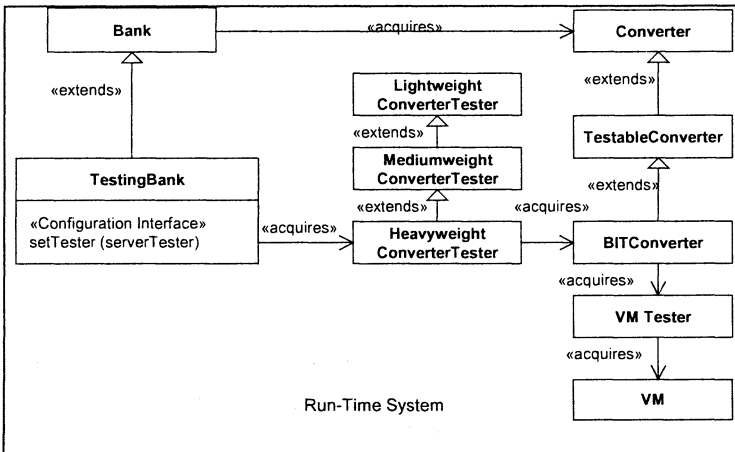


Figure 6. Test weight selection through a test configuration interface, and contract test of the underlying run-time system, for example a virtual machine (VM)

4.2 Explicit Versus Implicit Servers

In general, two kinds of server components can be distinguished. The first kind, known as explicit servers, correspond to the server components defined explicitly as part of the developed application. These represent server components as typically understood. The second kind, known as implicit servers, correspond to services supplied by the run-time system. This represents the run-time support for the features of the used language (e.g. I/O capabilities). With the advent of object-oriented languages and component technologies the trend is that more of the features that were traditionally embedded within the

run-time support software are now found in predefined library classes. Java provides a good example for this trend. The difference between these library features and the explicit server components of the first kind is that they are supplied implicitly.

A test of the run-time environment is performed when the component is executed on a new platform. Built-in contract testing provides a natural way to cope with this type of test. The approach for developing and deploying tester components is the same as for explicit server components. This is also illustrated in Figure 6. If a component comprises an in-built run-time system tester it may complain if it is executed in an unsuitable run-time environment.

4.3 Testing Deployed Component Instances

When a built-in test is applied to a component, the data stored within the component is changed. This is not a problem for server components which have just been initialized, but for already deployed components that have been executing for a while, the data stored in the component may be valuable. For example, the currency conversion component that is used in the bank context may contain a valuable list of exchange rates, and executing a built-in test on the converter would completely mess up these values. This clearly poses a problem because after the test, the component will be useless for the bank. The same problem arises if multiple clients access one single server component. This can be circumvented if an exact copy of the deployed instance is tested. Fortunately contemporary object oriented languages and some component technologies support such a mechanism (e.g. Java clone).

A similar problem arises, if a built-in reconfiguration test is performed in a system that must continue to provide its service. The new component that replaces an existing one may make the run-time system of the application fail during testing and thus jeopardize the entire system. We therefore recommend the use of different processes for executing built-in contract tests in a running system. The testing thread will notify the acquiring client when the test is successfully completed.

4.4 Support Framework for Built-in Contract Testing

Developing the contract testing artifacts for large and complex components is likely to demand considerable effort. This is natural, since bigger components also require more testing than smaller components, simply by the virtue of the fact that they provide more, or more complex, functionality. Built-in contract testing technology acknowledges this fact and supports the creation of the necessary artifacts through development frameworks. Such a support environment is currently available as a Java class library that supports

the development of built-in contract testing interfaces and components [3]. Its basic architecture is displayed in Figure 7.

The shaded box on the right hand side represents the contract testing artifacts that we have considered in this paper. The library simplifies the development of the testing interface quite considerably since it supports the implementation of the specification state model inside the testable component. It provides mechanisms that allow developers of the testing interface to define the states and transitions of the component. This reduces the testing interface to only two operations *isInState* and *setToState*, plus the feasible states that can be set and verified. The BIT state monitor that is provided with the library checks and sets the respective states and transitions during test. The test cases inside the tester component abide by the same fundamental interfaces as those of the library when they use the *setToState* and *isInState* operations of the state based testability contract.

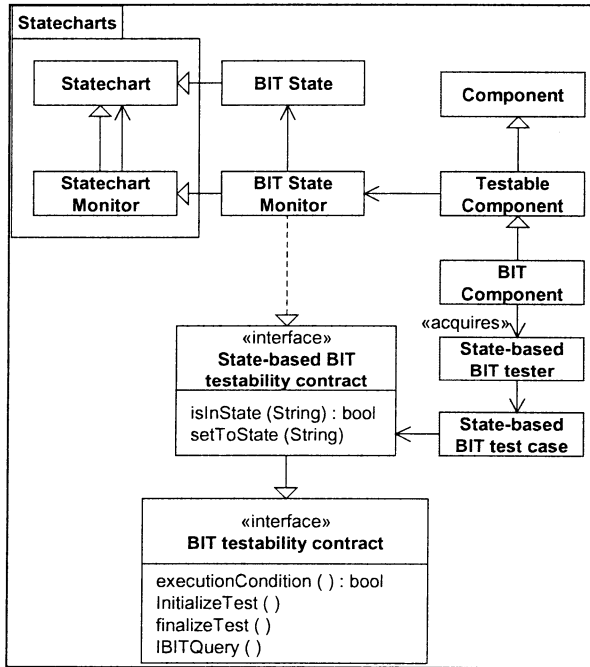


Figure 7. Architecture of a Java built-in contract testing support library

5. SUMMARY AND CONCLUSIONS

This chapter has described an approach for enriching components with the capability to verify their run-time integrity, in situ, by means of built-in tests. The idea of building tests into components is not new. However, previous approaches concentrate on built-in development time class testing [7], or have adopted a hardware analogy in which components have self-test functionality that can be invoked at run-time to ensure that they have not degraded [8]. Since software, by definition, cannot degrade, the portion of a self-test which rechecks already verified code is redundant, and simply consumes time and space resources. The approach described in this chapter augments the earlier work by focusing built-in test software on the aspects of a component's capabilities which are sensitive to change at run-time - namely the environment which provides the services used by the component. This new emphasis is characterized as built-in contract testing.

Built-in contract testing provides an architecture and a methodology that is particularly well suited for highly dynamic and distributed systems, such as Internet applications, and systems with dynamic reconfiguration. Since these are the applications primarily targeted by modern component technologies, built-in contract testing represents a natural extension to component-based software engineering practices.

The work is currently oriented towards reconfigurable information systems for which the overhead of run-time tests does not particularly affect efficiency considerations. Extending the technology to real-time and embedded systems still presents some challenges for future research in terms of how much in-built testing such systems may bear. We believe this methodology represents a contribution towards the practical applicability of component technology and component-based development practice. We have consequently integrated built-in contract testing into a general-purpose model-driven approach to component-based development, known as the Kobra method [2], which promotes the early design of tests along with functional artifacts. By extending the component paradigm with effective in-situ verification techniques and processes, built-in test technology brings the "plug and play" vision of component-based development one step closer to realization.

REFERENCES

- [1] Meyer, B.; Object-Oriented Software Construction. Prentice Hall, 1997.
- [2] Atkinson, C., et al.; Component-based Product Line with UML. Addison-Wesley, London, UK, 2001.
- [3] Component+ Project; Built-in Testing for Component-Based Development. Technical Report D.3, <http://www.component-plus.org>, 2001.
- [4] Binder, R.V.; Testing Object-Oriented Systems. Addison-Wesley, Reading MA, 1999.

- [5] Pressman, R.S.; Software Engineering, a practioner's approach, McGraw-Hill, New York, 1997.
- [6] Somerville, I.; Software Engineering. Addison-Wesley, Reading MA, 1995.
- [7] Jézéquel, J.-M., Deveaux, D., and Le Traon, Y.; Reliable Objects: Lightweight Testing for OO Languages. IEEE Software, July/August, 2001.
- [8] Wang, Y., and others; Built-in Test Reuse in Object-Oriented Framework Design. ACM Journal on Computing Surveys, 23(1), March 2000.

ACKNOWLEDGEMENTS

This work is partially funded by the EC IST 5th Framework Project, Component+. The contribution of the Component+ consortium is gratefully acknowledged.

APPENDIX

Table 1. Converter component state transition table

#	initial state	precondition	event	final state	post condition
1	empty		convertFromEuros (Curr,Amount)	empty	exception
2	empty		convertToEuros (Curr,Amount)	empty	exception
3	empty		setRate (Curr,Rate)	loaded	(Curr,Rate) stored in Conv
4	loaded	[Curr in Conv]	convertFromEuros (Curr,Amount)	loaded	return conversion
5	loaded	[Curr in Conv]	convertToEuros (Curr,Amount)	loaded	return conversion
6	loaded	[Curr ! in Conv]	convertFromEuros (Curr,Amount)	loaded	exception
7	loaded	[Curr ! in Conv]	convertToEuros (Curr,Amount)	loaded	exception
8	loaded	[n<max-1 & Curr in Conv]	setRate (Curr,Rate)	loaded	update (Curr,Rate) in Conv
9	loaded	[n<max-1 & Curr ! in Conv]	setRate (Curr,Rate)	loaded	(Curr,Rate) stored in Conv
10	loaded	[n = max - 1 & Curr ! in Conv]	setRate (Curr,Rate)	full	(Curr,Rate) stored in Conv
11	full	[Curr in Conv]	convertFromEuros (Curr,Amount)	full	return conversion

12	full	[Curr in Conv]	convertToEuros (Curr,Amount)	full	return conversion
13	full	[Curr ! in Conv]	convertFromEuros (Curr,Amount)	full	exception
14	full	[Curr ! in Conv]	convertToEuros (Curr,Amount)	full	exception
15	full		setRate (Curr,Rate)	full	exception

Table 2. Specification of the state verification and state setting operations of Converter's testing interface

testing interface – state verification operations		
Boolean	isEmpty ()	Returns true if Conv is in the empty state. n = 0
Boolean	isLoading ()	Returns true if Conv is in loaded state. n = 1 .. max -1
Boolean	isFull ()	Returns true if Conv is in the full state. n = max
Boolean	isEntry (Curr)	Returns true if Curr is stored in the Converter.
Boolean	isEntry (Curr, Rate)	Returns true if the pair (Curr,Rate) is stored in C.
Boolean	ruOk ()	check of all internal assertions
testing interface – optional state setting operations		
void	setEmpty ()	sets the Conv into empty state. n = 0
void	setLoaded (n, {Curr,Amount}[])	sets the Conv into loaded state. n = 1 .. max -1
void	setFull ({Curr,Amount}[])	Returns true if Conv is in the full state. n = max
void	setEntry (Curr)	sets an entry Curr
void	setEntry (Curr, Rate)	sets an entry (Curr,Rate)

Chapter 5

Interfaces and Techniques for Runtime Testing of Component-Based Systems

Jonathan Vincent¹, Graham King¹, Peter Lay² and John Kinghorn²

¹*Intelligent Systems Laboratory, Southampton Institute, UK;* ²*Philips Semiconductors, Southampton, UK*

Abstract: Code reuse has always been an important area for reducing development time. The component-based software development paradigm is becoming increasingly important as a structured way of achieving effective code reuse, moving away from the ad-hoc processes that have tended to be used in the past. However, attention must be given to difficulties associated with testing a system composed of (possibly third party) components whose internal structure is not visible to the system integrator. In general, it cannot be assumed that a component will function as expected when exposed to a specific usage profile in a particular target environment, which will usually differ from that used by the component developer for testing purposes. It is unrealistic to expect non-trivial components to be 100% defect free. A further complication arises when components, possibly of different manufacture, must interact to meet the overall requirements of the software system. To facilitate the construction of robust component-based software systems, it is argued that a software component must provide some form of test service that enables the verification of correct internal functioning in the target environment, and supports the verification of component interactions. This chapter describes an architecture that facilitates such tests, and briefly outlines some of the test types that may be incorporated into the component.

Key words: Software components, software test, built-in test

1. INTRODUCTION

This chapter examines an architecture for Built-In-Test for Run-Time-Testability (BIT-RTT) in software components, expanding and extending previous work by Wang *et al.* [1], with specific attention paid to continuous

test and real-time systems. The current trend of increased usage of COTS (Commercial Off-The-Shelf) components [2], typically supplied by component vendors as binaries, imposes certain restrictions and difficulties on the testing and verification of system correctness, because the internal mechanisms of some or all system components will be unknown. Components do not generally provide test facilities, and there are no standardised built-in tests. The testability of systems constructed with COTS components is therefore low, and whilst there are numerous approaches to improving the reliability of a software product (including, amongst others, inspections and formal methods), it is axiomatic that reliability is enhanced by appropriate testing.

BIT seeks to address some of the difficulties associated with component-based development, by extending the traditional view of a component [3] to incorporate techniques which facilitate test and verification when integrated within a system, thereby raising the test visibility of COTS components. There are two principal motivations for considering the use of BIT mechanisms. First, with any non-trivial component there will be some non-zero probability of it containing residual defects not detected during the component vendor's testing cycle. Second, a software system may rely on the correct interaction of many COTS components, and issues of correct component usage, dynamic behaviour and environmental considerations need to be verified by the system integrator. The inclusion of testing facilities within the component itself supports the detection and localisation of residual defects which only manifest themselves at integration or deployment time, and assist the system integrator in verifying correct component interactions. Since the occurrence of error events typically depends upon the usage profile, and usage varies with time, continuous verification of component and system behaviour is advocated.

This chapter focuses on the architecture and primary interfaces of the BIT approach. A discussion of the philosophy and implications for product quality may be found in [4] and [5].

2. TESTABLE COMPONENTS

A component is defined by a number of provided and required interfaces, through which its functionality is understood. The provided interfaces allow external entities (clients) to access the services provided by the component, whilst the required interfaces define services which the component needs in order to function correctly. A standard component is therefore constructed from a number of what will be referred to here as functional interfaces. In general, a system developer is unable to look inside a component (and ideally should not wish to do so). However, problems can arise in a system composed

of COTS components when there is no provision for testing. The system developer must be able to verify that the components used function correctly in the specific target environment that the system is intended for. This is likely to differ from the component developer's test environment(s). It must also be verified that components interact correctly in order to meet the overall system requirements. In particular, issues related to resource utilisation (memory allocation, deadlock, etc.) can be complicated by the use of black-box components.

A BIT-component is composed of its normal functional interface(s), augmented by one or more test interfaces (see Figure 1). The component developer provides these interfaces to access built-in test services which can support integration test and continuous verification.

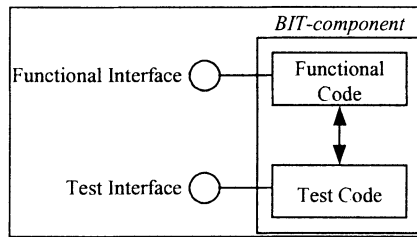


Figure 1. Concept of a BIT component

Errors occurring in a component based system can be classified at two levels; those that are confined to a specific component and can be detected by built-in tests within that component, and those at system (or sub-system) level arising from incorrect component interaction and cannot be detected within the components. For example, data integrity can be verified within a component, but deadlock, which is an example of a system level problem caused by resource competition, cannot. Verification of system level behaviours therefore requires external test components. The BIT interfaces provide for both internal and external tests; the former by building tests into the component, the latter by enabling information not normally visible to be safely accessed by the external test components.

3. BIT ARCHITECTURE

The BIT architecture is based on the following architectural elements:

- a) *BIT-components*: components which provide a number of built-in test services.
- b) *Testers*: components which use the test services of BIT-components to determine whether a system-level error condition exists.

- c) *Handlers*: components which handle errors detected by BIT-components or test components.
- d) *Constructors*: a conceptual element, nominally responsible for the instantiation of (high level) BIT-components, testers, and handlers, and their interconnection.

The BIT architecture specifies a number of interfaces which BIT-components, testers, handlers and constructors must provide. In the case of handlers and constructors, however, much is left to the system designer as error processing is application specific and the method of system construction (i.e. component instantiation and association) varies. The BIT approach does not advocate the use of any particular component architecture, nor is it confined to any one implementation language. Only those aspects which must be specified for compatibility are specified, leaving sufficient flexibility and extensibility for wide application. This typically means the definition of public interfaces, data types, and constants.

In general, information from the component can flow in two ways. First, an external entity can access the information via functions of the appropriate BIT interface, in a polling mode. Second, a callback mechanism can be established, where appropriate, so that external entities are notified when events of interest occur. At least one, and possibly both, modes of operation are supported by each BIT interface. By convention, interfaces to test services provided by a component are named *IBITx*, where *x* is the type of test catered for. Where an external tester is involved, and communication is required from component to tester, the corresponding interface on the tester is named *IBITxNotify*. As an example, to facilitate deadlock testing, components may provide an *IBITDeadlock* interface. A deadlock tester provides the *IBITDeadlockNotify* interface, through which a component can notify the tester of events of interest, in this case resource requests and releases (see Figure 2). Similar relationships may exist between, for example, handlers and BIT-components, and handlers and testers. There can be any number of test interfaces, to address a wide variety of problems, including, but not limited to; deadlock testing, timing verification and performance profiling, memory management, code integrity, data integrity, and trace facilities.

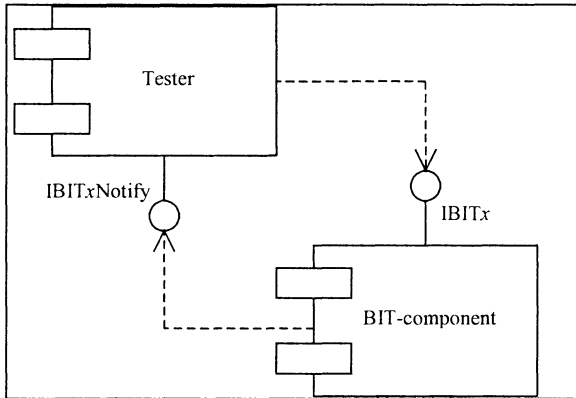


Figure 2. IBITx and IBITxNotify interfaces

Whilst any number of test interfaces may be specified, four "primary" interfaces are defined which form the foundation of the architecture. The primary interfaces are:

- IBITQuery*: allows an external entity to query the availability of specific test services and, if present, to acquire a handle to the corresponding *IBITx* interface.
- IBITError*: provides the means for error propagation. The *IBITError* interface can be queried to determine the error status, or can be configured to notify a corresponding *IBITErrorNotify* interface of the occurrence of specified error events.
- IBITErrorNotify*: provides a mechanism for error notification, as typically provided by handlers so that they can be informed by BIT-components and testers when specified error events occur.
- IBITRegister*: provides the mechanism for associating BIT-components with testers and handlers. The interface provides functions for notifying the creation and destruction of BIT-components. Through this interface, a system specific "constructor" can be created for the instantiation and connection of BIT-components.

The full specification of these interfaces will not be given here but the main concepts and important supporting functions will be discussed. *Table 1* defines which interfaces are mandatory on which architectural elements. Note that, since the *IBITError* interface supports polling as well as a callback mechanism, *IBITErrorNotify* is not mandatory, but would typically be provided by most handlers. Note also that a BIT-component is defined as such by the provision of the *IBITQuery* interface - it does not have to provide any other test facilities. Thus, compatibility with the BIT architecture is easily achieved. The provision of a pair of corresponding interfaces, such as *IBITError* and *IBITErrorNotify*, would be typical in a BIT environment, as

some flexibility is necessary in the way in which information is distributed. Support for both polling and callback modes of operation (i.e. pulling and pushing of data) offers the flexibility required to meet the demands of a wide spectrum of application areas.

Table 1. Mandatory BIT interfaces

Element	IBITQuery	IBITError	IBITErrorNotify	IBITRegister
BIT-component	✓			
Tester	✓	✓		✓
Handler	✓			✓
Constructor				✓

3.1 BIT components

A BIT-component provides one or more test interfaces which allow access to built-in test services. Test services are grouped into logical blocks, such as data integrity test, code integrity tests, trace services, timing test, deadlock test, etc. Such services are an optional part of a BIT-component. All BIT-components have a query interface, referred to as *IBITQuery*, which allows handlers, testers, and other external entities to determine which, if any, BIT services are provided by the component. The main provision of *IBITQuery*, is a function named *IBITQuery :: QueryInterface*, which allows the existence of a particular BIT interface to be ascertained and, if present, the corresponding handle to be acquired.

Typically, however, a BIT-component will include BIT services. If internal error detection mechanisms are included, the BIT-component will provide an *IBITError* interface, which enables errors to be propagated to system level components with the responsibility for error handling. Two modes of operation are supported by *IBITError*; polling and callback. Using the *IBITError :: QueryErrorStatus* function an external entity can determine whether any errors have been detected within the component and of what type. Alternatively, using the *IBITError :: SetHandler* function, a link can be made between the *IBITError* interface and a corresponding *IBITErrorNotify* interface (as typically provided by handlers). When an error of interest occurs, the BIT-component informs the handler by calling its *IBITErrorNotify :: ErrorNotify* function. In general, a BIT-component may be represented as in Figure 3. Note that the BIT approach does not alter in any way the component's "functional interface".

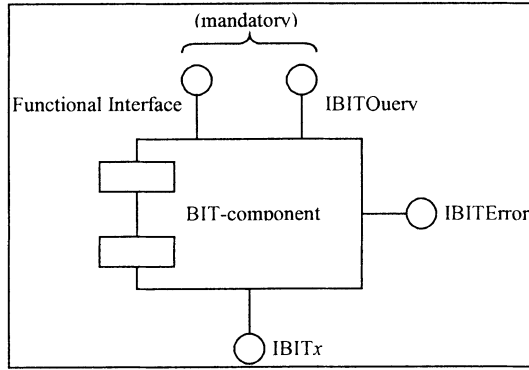


Figure 3. General structure of a BIT-component

3.2 Testers

Testers are separate components that are used to verify higher level behaviours, for example, deadlock conditions, which arise from component interaction and cannot therefore be verified within the components. Testers interface with BIT-components, gaining handles to available *IBITx* interfaces (i.e. interfaces to test services), via the *IBITQueryInterface* which is provided by all BIT-components. Like BIT-components, testers must provide an *IBITQueryInterface* through which its other interfaces may be accessed. Testers must also provide an *IBITRegister* interface. This defines two functions; *IBITRegister :: NotifyCreation* and *IBITRegister :: NotifyDestruction* which are used to inform the tester of the creation of new BIT-components and the destruction of existing BIT-components. A handle to the corresponding BIT-component is passed as a parameter to these functions. The tester can then access the *IBITQuery* interface of a registered component to ascertain the availability of test services and configure them as required (see section 3.4).

Where a BIT-component is required to notify a tester of the occurrence of particular test events, the tester will provide the BIT-component (via the *IBITx* interface) with a handle to the corresponding *IBITxNotify* interface. An *IBITxNotify* interface is not specified if the *IBITx* interface only supports polling for test information. If the *IBITx* interface supports both polling and callback modes of operation (such as the *IBITError* interface) a tester may or may not then choose to include an *IBITxNotify* interface, at the discretion of the system designer. *IBITxNotify* becomes a requirement when the only supported mode is callback. Like BIT-components, a tester propagates errors it detects to handlers or other interested parties via its *IBITError* interface, which is a compulsory provision. Testers may, of course, have other

application specific interfaces that are not specified here. In general, a tester component may be represented as in Figure 4.

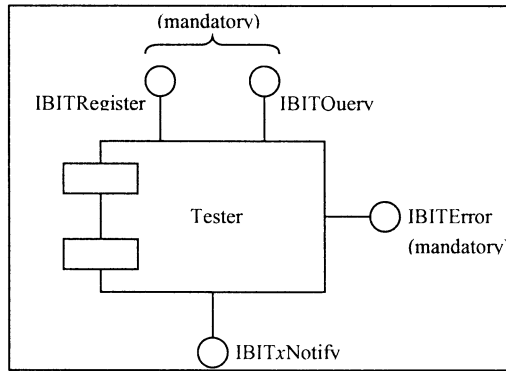


Figure 4. Structure of a tester

3.3 Handlers

A handler is a component with the responsibility for error processing. A system may have any number of handlers dedicated to processing specific error types, although one handler per system would be typical. Error processing is not defined as part of the BIT approach, being considered application specific. A handler provides three specified interfaces; optionally *IBITErrorNotify* (if it wishes to use the callback mechanism of the *IBITError* interface of associated BIT-components and testers) and compulsorily *IBITQuery* and *IBITRegister*. The latter provides two functions; *IBITRegister :: NotifyCreation* and *IBITRegister :: NotifyDestruction* which are used to inform the handler of the creation of new BIT-components and the destruction of existing BIT-components. The handler can then access the *IBITQuery* interface of a registered component to ascertain the availability of test services, configure these services, and configure error notification. In general, a handler component may be represented as in Figure 5.

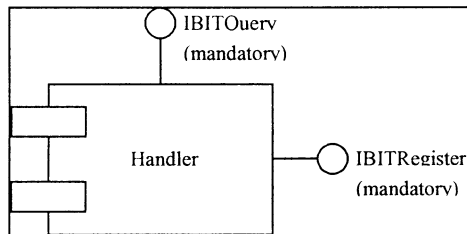


Figure 5. Structure of a handler

3.4 Constructor

The constructor is a conceptual entity and its general interfaces and implementation are not specified as this may lead to a reduction in the scope of applicability of the BIT technology. The constructor is nominally responsible for notifying the testers and handlers of the creation and destruction of BIT-components via their *IBITRegister* interface. In a simple system this action might be hard-coded to occur at start-up. In a more complex system this responsibility may require some centralised "component factory" or may be distributed throughout. The method of system initialisation and component instantiation is not prescribed here.

An important provision is made to support the development of a component factory should an application require it. It cannot be assumed that component instantiation is a high level activity; a system is generally constructed in a hierarchical way, and a BIT-component may instantiate child components. It is necessary to ensure that the system becomes aware of the creation of these components. Therefore, when a BIT-component is created, it is passed a handle to an *IBITRegister* interface. This handle will be stored by the component and passed on to any child components. On creation, the BIT-component uses this handle to call the *IBITRegister :: NotifyCreation* function. The BIT-component will not be aware of the owner of this interface - it could be a component factory, a tester, or a handler. The passing of this handle is obligatory, but if the system designer does not wish to utilise this mechanism, then a null handle can be provided instead.

As an example of how this facilitates component registration, consider the following possible sequence of events. On start-up, the main thread of the application instantiates a constructor component, which in turn creates the necessary handlers and testers, maintaining handles to them in an internal table. The constructor registers the testers with the handlers so that error conditions can be propagated. The main thread then creates some initial BIT-components, passing each a handle to the constructor component's *IBITRegister* interface. On creation, they call the *IBITRegister :: NotifyCreation* function and the constructor can then register them with the appropriate testers and handlers listed in its internal table. The testers and handlers receive notification of the creation of a new BIT-component and can then determine what test services are offered by the BIT-component via the *IBITQuery* interface. Child components can then be safely created, and will be automatically registered with the required testers and handlers. Figure 7 in Section 4, illustrates this sequence of events. In simpler systems, where there is, for example, just a handler and no external testers, the same mechanism can be used to automate the registration of BIT-components with the handler, by passing a handle to the handler's *IBITRegister* interface when BIT-components are created.

4. EXAMPLE CONFIGURATION

Figure 6 illustrates an example BIT system configuration comprising three BIT-components (one of which is a child of a higher level BIT-component) with deadlock testing support, an external deadlock tester, a handler, and a constructor, showing the interfaces involved (not all associations are shown for clarity).

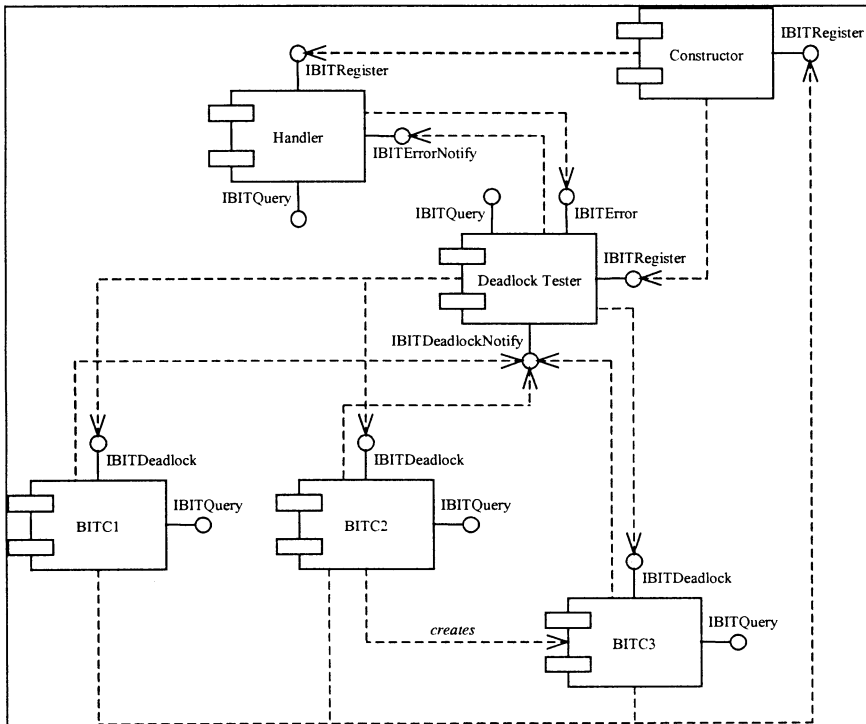


Figure 6. Example system configuration

A representative series of component interactions is illustrated in Figure 7. In this example, the constructor is implemented such that it creates the system's initial (top-level) components, including handlers and testers. It maintains an internal table of handles to the *IBITRegister* interfaces of handlers and testers so that they can be notified of the creation of new BIT-components.

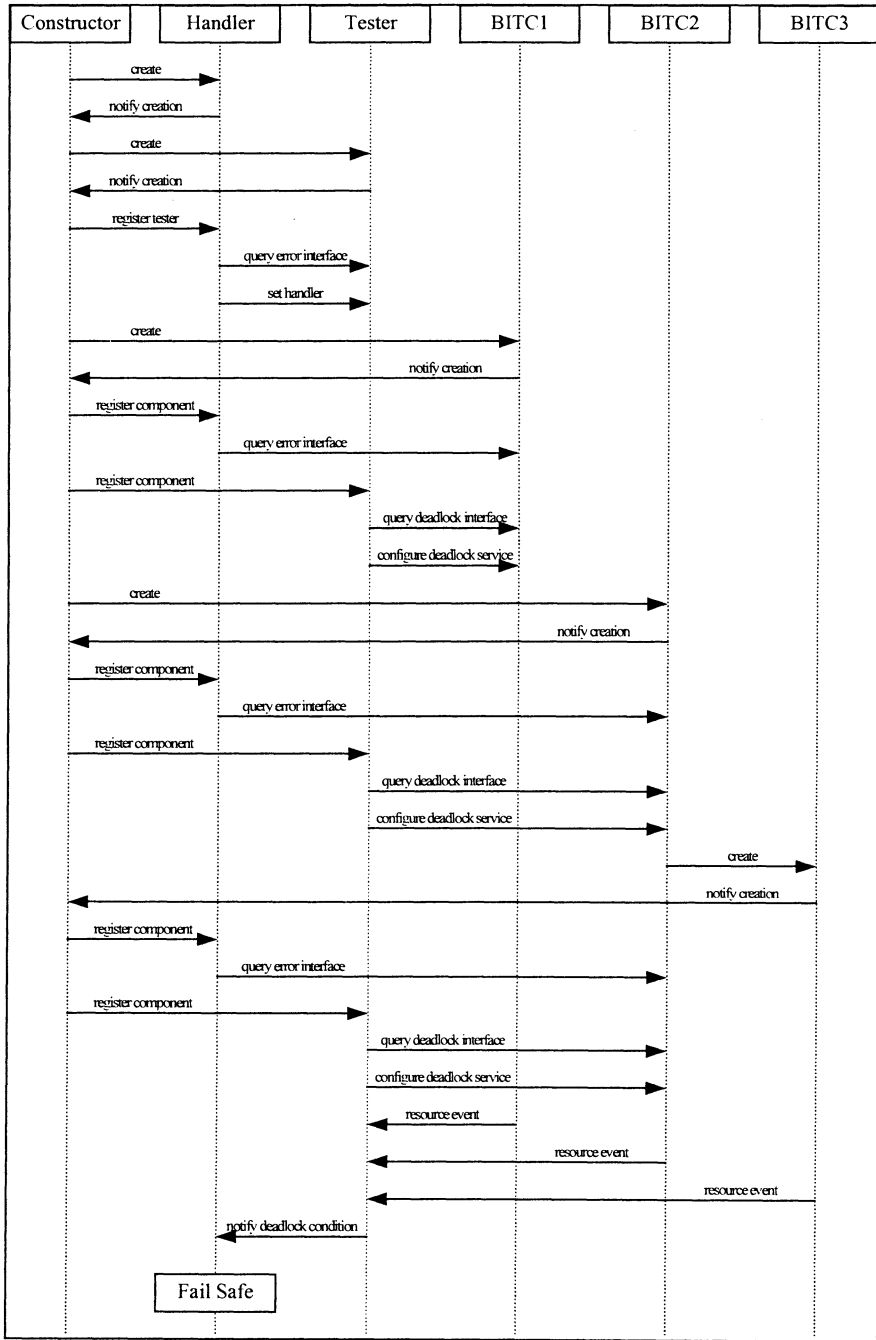


Figure 7. Sequence diagram of typical component interactions

On start-up, the constructor first creates the handler, passing it a handle to the constructor's *IBITRegister* interface. The constructor is then called back using the *IBITRegister :: NotifyCreation* method which is passed a handle to the handler's *IBITQueryInterface*. This handle is stored by the constructor in an internal table. The constructor then creates the deadlock tester in a similar way, and when called back, it notifies the handler of the tester's creation. The handler queries the tester for the presence of an *IBITError* interface (which it has, being a compulsory provision on testers). Having acquired a handle to this interface, the handler is able to configure it, using the *IBITError :: SetHandler* function, so that the tester will notify the handler of error conditions.

The constructor then creates BIT-components *BITC1* and *BITC2*, again passing a handle to its *IBITRegister* interface. The handle is stored by each BIT-component for future use. The constructor is then called back by the BIT-components, which then notifies the handler and tester of their creation (it could have done this directly, but this provision is made primarily for components lower in the hierarchy). In this example, the BIT-components only support deadlock testing and consequently have no *IBITError* interface. The handler therefore ignores them. The tester, however, queries their services via the *IBITQuery* interface and confirms the presence of *IBITDeadlock* interfaces (the only interface of interest to this particular type of tester). Their deadlock interfaces are then configured to notify the tester of resource related events, such as resource allocation and release, via the tester's *IBITDeadlockNotify* interface.

Whilst the communication between the constructor and the initial (high-level) components that it creates might at first seem excessive or unnecessary, using this generalised method of component creation automates the registration process, ensuring that all system level components are aware of the existence of new BIT-components not created directly by the constructor. In this example, *BITC2* creates a child component, *BITC3*, passing the handle stored when *BITC2* was created. *BITC3* uses this to notify the constructor of its creation, which then proceeds, as before, to notify the tester and handler of the existence of this new component. In this example, the handler does not find an *IBITError* interface, but the tester finds an *IBITDeadlock* interface and configures it as previously described.

During execution, resource related events are notified to the deadlock tester, which responds to each event by constructing and reducing a Resource Allocation Graph [6] to determine whether a deadlock condition exists. When this occurs, the handler is notified of the error via the *IBITErrorNotify* interface. The handler can then take whatever application specific error processing is required. This example shows a number of resource notifications, and the subsequent identification and handling of an error event by fail safe.

Although not shown in this example, destruction of a component must be handled properly. Specifically, a component cannot be destroyed before all handles to it are freed. The BIT architecture does not specify the mechanism for this, as it may be specified by the component model. The *IUnknown* interface in Microsoft's Component Object Model (COM), for example, uses the *AddRef* and *Release* methods to maintain an internal count of the number of references to its interfaces. When this count reaches zero, the component is destroyed. The BIT architecture provides the *IBITRegister* :: *NotifyDestruction* method to assist in safe termination. When a constructor, handler or tester receives this notification, all internal references to the specified component must be deleted.

5. TEST SERVICES

The previous sections have described the BIT architecture, mechanisms for construction and error propagation, and briefly outlined the four primary interfaces. In this section, a number of possible test services are outlined. The provision of an *IBITError* interface enables errors detected within a BIT-component to be propagated to a handler or other interested component. The component developer may simply instrument the functional code of the component with various tests for, for example, pointer validation, correct completion of operations, I/O errors, etc. However, it is more convenient from a reuse point-of-view if such tests are logically grouped into services that can be included in the BIT-component as required. Testers/handlers would gain access to these services via the *IBITQuery* :: *QueryInterface* function, whilst internally these services can provide helper functions to the component to speed development.

5.1 User conformance testing

User Conformance Testing is concerned with verifying that the system is using the component appropriately. This involves the validation of parameters for API errors, to ensure that requested operations are valid in the context of the current component state.

5.2 Code integrity

Software is static in nature – once compiled a software component cannot change. However, software can be corrupted in a system due to a variety of reasons such as copying errors when the software is loaded into RAM for execution (note that although many embedded systems traditionally execute from ROM, many now execute from RAM for speed reasons) or corruption

due to programming and data errors. Many systems do not employ virtual memory where processes are kept physically separate. Therefore, software can be corrupted by other processes manipulating memory outside of their allocated address space. Error detection codes such as a checksum or CRC (Cyclic Redundancy Code) can be used to verify the code integrity at run-time. Such verification could be invoked periodically by the component itself, or instigated by an external entity via the corresponding test interface. In practice, this facility is likely to be most appropriate for embedded systems.

5.3 Data integrity

Two approaches to data integrity are considered. First, a checksum (or other error detection code) can be used to determine whether a set of data items has been corrupted by unauthorised access of the component's data space. On initialisation, the checksum of the data items is computed and stored. After each genuine operation on this data, the checksum is recomputed and the stored value updated. At any time, the integrity of the data can be verified by recomputing the checksum and comparing with the stored value.

Second, it is often the case that some relationship between data items can be defined and verified. This relationship is dependent on the specification and implementation of the component and can only be specified by the component developer. Examples include, checking correct linkage in linked lists, verifying that node data meets the constraints in binary trees, verifying that the result of a sort operation is sorted data, and verifying the consistency of internal state machines.

5.4 Deadlock

Deadlock is a system level problem arising from resource competition and mutual exclusion. Detection of a deadlock condition necessitates the use of an external deadlock tester, as previously described. The tester, when informed of the creation of a BIT-component, ascertains whether that component can provide resource and thread information. Assuming it can, the deadlock interface is configured to notify the tester of resource related events. The tester can determine whether a deadlock condition exists by constructing and reducing a Resource Allocation Graph.

5.5 Timing

In real-time systems, the correctness of a computation depends not only upon the value of the result, but also its timeliness. Even in non-real-time systems, there are often overall quality-of-service levels that must be met. Whilst it is possible, under certain restrictive conditions, to analytically

determine system schedulability (using, for example, Rate Monotonic Analysis [7]), a system constructed from COTS software components is unlikely to provide sufficient information to do so. Further, the results of such analysis are often too conservative for practical use. Under these circumstances, it is desirable to build-in support for timing verification.

Note that the knowledge of deadlines may reside at one of two levels; internally, the component may have certain intrinsic timing requirements of which the system integrator is not fully aware, whilst many time critical processing tasks can be expected to involve many components, and therefore the deadline is known only at system level. The use of a timing tester component may therefore be required to handle notification of event start and finish times, verification of results, and notification of discrepancies to the handler. Support for timing test is also an important part of the code optimisation process.

5.6 Detection of residual defects

In non-trivial components it is not generally possible to verify that the code is 100% defect free. Typical causes of problems that show up after extended periods of execution include invalid pointers, memory allocation / deallocation, arithmetic errors (including accumulated floating-point errors), out of bounds array indices, corrupt data structures, errors in untested or untestable code, exit conditions of recursive algorithms, type conversions, unexpected/untested input, logical errors (especially in conditional expressions), timing errors, and optimisation errors. Such defects may cause anything from complete system failure to minor data loss or performance degradation, and the defect may not appear as a fault for some considerable time after the initiating event. In order to improve the quality and maintainability of software systems and the contained components, it is necessary to improve the detection rate of residual defects, and provide sufficient diagnostic output to enable remedial action to be taken. Residual defects are generally detected by the inclusion of assertions and consistency checks within the normal executable code of the component, providing continuous verification of program behaviour. Such checks examine, for example, consistency of component state and data structures, numeric results, program output, bounds, etc. Analysing design models, such as state-transition diagrams, will inform the design of suitable tests.

5.7 Trace

Incorporating BIT services into a software component is expected to improve the error detection rate of both component and system. However, knowing that an error has occurred is not particularly helpful from a

maintenance perspective. Including tracing facilities within the component allows information about, for example, operations invoked, values of data items, messages received, interrupt events, etc. to be recorded for later use in assisting the localisation of a defect. This is a similar provision to that provided by many development environments, with the exception that this trace is permanently available within the component, to be enabled and disabled as the situation requires. It can be used during system test or can be enabled throughout the component/system lifetime. Typically, trace messages will be stored in a fixed size buffer, to be recorded onto permanent storage when an error condition of interest has occurred. In effect, this is the software equivalent of a logic analyser.

6. CONCLUSIONS

This chapter has outlined a generic architecture for implementing built-in-test in software components, as an extension to traditional component technology. The architecture provides a flexible method of system configuration and error propagation within a component based environment. Component based development, whilst having great potential for reuse, is complicated by the lack of test visibility currently provided. It is suggested that the approach described can address many of the difficulties associated with component-based development, which arise from encapsulation and information hiding.

With the continuing expansion of the component-based software development paradigm, raising test visibility and confidence in vendors' components is of increasing importance. Whilst it is generally accepted that unit testing is a particularly cost effective stage at which to test traditional software, due to the potentially significant differences between a vendor's test environment / usage profile and that of the user, and the necessity to carefully examine component interactions, the component-based paradigm requires that testing effort be partially relocated towards the software integration phase. The development of a BIT technology, which can be readily incorporated into COTS components, enables test visibility to be raised. It is suggested that this approach may provide the basis for the construction of more testable, reliable, maintainable and generally higher quality component-based software systems, offering potential benefits to component vendors, system integrators, and end-customers alike. The standardisation of the architecture and interfaces is seen as a key factor in enabling built-in-test within the COTS market, and is an area of ongoing work.

ACKNOWLEDGEMENTS

This work is partially funded by the EC IST 5th Framework Project, Component+. The contribution of the Component+ consortium is gratefully acknowledged.

FURTHER INFORMATION

Further information, including the latest documentation, can be found at the Component+ web site: www.component-plus.org

REFERENCES

- [1] Wang Y, King G, Patel D, Court I, Staples G, Ross M, Patel S, On built-in test and reuse in object-oriented programming, ACM Software Engineering Notes, 23(4), pp60-64.
- [2] Arif Ghafoor S, Paul R, Software engineering metrics for COTS-based systems, IEEE Computer, May 2001, pp91-95.
- [3] Szyperski C, Component software beyond object-oriented programming, Addison-Wesley, 1998.
- [4] Vincent J, King G, Built-In-Test for Run-Time-Testability in Software Components: Testing Architecture, Software Quality Management, British Computer Society, 2002.
- [5] Vincent J, King G, Built-In-Test for Run-Time-Testability in Software Components: Product Quality Implications, Software Quality Management, British Computer Society, 2002.
- [6] Bacon J, Concurrent systems, operating systems, database and distributed systems: an integrated approach, 2nd edition, Addison-Wesley, 1998.
- [7] Liu C, Layland J, Scheduling algorithms for multiprogramming in a hard real-time environment, Journal of the Association for Computing Machinery, 1973.

Chapter 6

The NEPTUNE Technology to Verify and to Document Software Components

Juan Carlos Cruellas¹, Jean-Paul Bodeveix², Thierry Millan² and Agusti Canals³

¹UPC, Spain; ²IRIT, Toulouse, France; ³CS, Toulouse, France

Abstract: The main objective of the NEPTUNE project (Nice Environment with a Process and Tools Using Norms and Example) is to develop both a method and tools (complementary to the existing software environments) based on the use of the UML notation. This method, gained from considerable experience in the industrial environment, will apply to a variety of different fields: software development, business processes and knowledge management. The newly developed tools will enable static check of UML models for their coherence. They will also enable generation of professional documentation resulting from the transformation of models. This will be compliant with the context of the UML notation and will take into account user's requirements. The method and tools developed in this way will facilitate the application of the UML standard as well as promoting its use in a large number of varied fields.

Key words: Methodology, rules checking, , documentation, tools, UML

1. INTRODUCTION

The UML notation supports a great part of the lifetime of software components or applications starting from requirements analysis down to the detailed design. It is object-oriented from the preliminary phases of the development and thus departs from classical design and analysis methods.

UML defines the semantics and the notation of the concepts used for software components design and component interaction, the way in which these concepts can be applied, the views and document types that can be

realized. A standardized methodology for UML does not exist. However, different methodologies are used in the UML context: Catalysis, Unified Process, Objectory, *etc.*

After its adoption as a standard, UML is increasingly used for the construction of design and analysis models of software systems, business processes, knowledge engineering, *etc.* Application domains being increasingly multidisciplinary, each partner of a design team describes its own view of the components. The UML notation, through its support for multiple views, helps in this collaboration. It is also justified by the ever-increasing number of models built using UML and by the reusability extension throughout all levels: patterns, architectures, models, and components.

Although the formalisms used in UML support model checking better than the formalisms used in any other heuristic design and analysis methods, neither UML, nor the methodologies describing their use refer to this possibility.

Currently, the verifications mentioned above are performed at the level of development environments, during code compilation, and application debugging and testing, that is to say in the last stages of software development. This situation exhibits at least two major drawbacks: propagation of errors introduced in the early development stages and impossibility of checking models for which there is no code generation such as organizational models, for example.

The aim of the NEPTUNE project is to provide process guidelines for building software components, and tools for checking UML diagrams and generating documentation. The process guidelines are dedicated to different business lines. The objective is to provide the users with guidelines that facilitate the management of projects with UML. The tool for checking UML diagrams supports co-operative design in different environments, and ensures coherence of multiple views. The tool for generating documentation gives the capability to generate information and to organize this information from different points of views, which complement the different diagrams already designed in the case tools.

2. NEPTUNE ARCHITECTURE

The NEPTUNE architecture is based on the use of three standardized technologies: XMI (XML Metadata Interchange standardized by OMG), XSL (eXtensible Stylesheet Language) and OCL (Object Constraint Language). XMI is a format depicted in XML (eXtensible Markup Language) applied to the UML meta-model to allow interchange of models. The different UML case-tool editors currently support this format. XSL is a powerful language to manage structured information expressed in XML. This language is used to

construct style sheets containing both formatting vocabulary, such as HTML, and expressions of structured language. Applying XSL style sheets on different XMI expressions of UML models gives the user the capability to obtain well-defined documentation. OCL is the standardized part of the UML language devoted to the expression of constraints on UML class schemas. This language permits the expression of properties, which cannot be represented graphically.

Three modules compose the NEPTUNE architecture. The first one transforms a XMI file of the UML representation into an internal form. An API provides the operations necessary to access the different elements of the internal form. The aim of this API and of the internal representation is to provide the functionality necessary for the checker and the document generator, regardless of the tool used for the edition of UML diagrams. The two other modules are the checker and the document generator. The checker applies rules on the UML model within the internal form. These rules are expressed in OCL+ and are applied at the meta-model level. OCL+ is an extension of OCL including temporal logic operators and transitive closure computation. The document generator uses the XSL language to extract and transform the information presented in the UML model into information readable by people not familiar with UML notation. Figure 1 summarizes the global architecture of the NEPTUNE tools specifying the inputs/outputs. In particular, it shows that the API loads the XMI file and provides the UML elements required by the checker and the document generator. The checker provides warning and error messages as an output. The documentation generator produces XML, HTML and PDF documents.

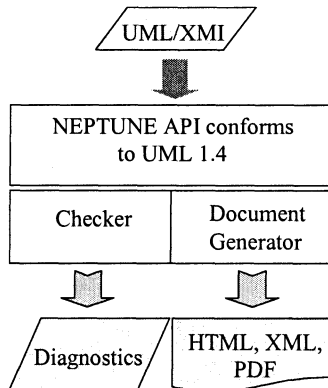


Figure 1. Global architecture of Neptune

3. OVERVIEW OF THE NEPTUNE METHOD

The UML/NEPTUNE process provides guidelines for Business Process, Knowledge Management and Software Engineering modeling. This process is divided in phases, and each phase is composed of activities. In this section, we will present an overview of the various phases for the Software Engineering domain (see Figure 2).

The Unified Process proposed by the original designers of UML includes a "Design and analysis" part. Our process is an operational instance of this part for the industrial context. The process is naturally iterative, but we choose to represent it sequentially for easier reading.

This process proposes packages and diagrams for each phase (see the browser architecture).

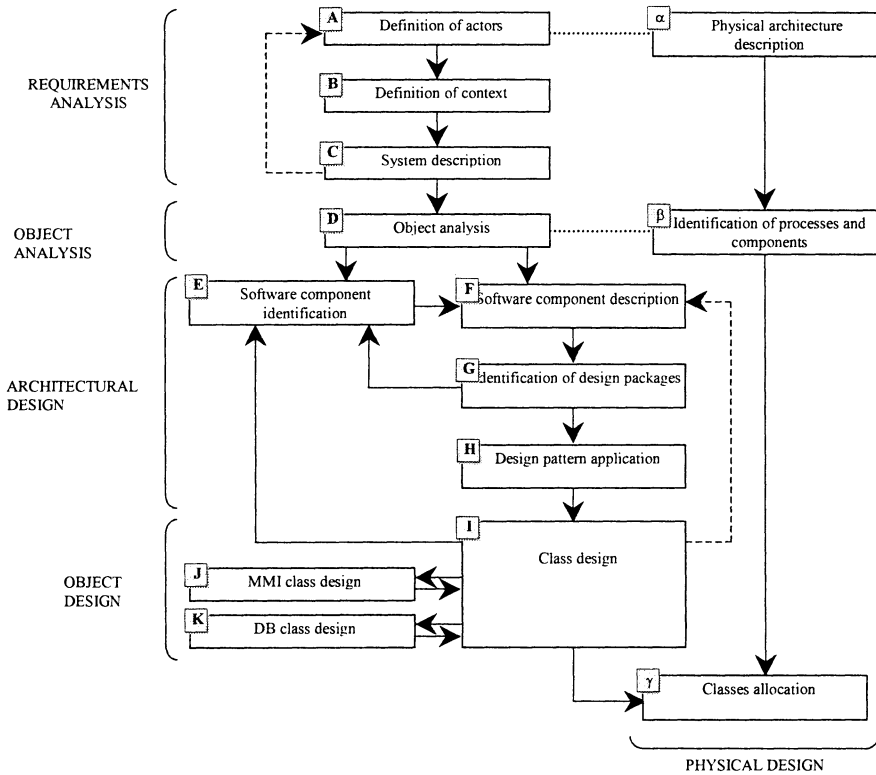


Figure 2. Software engineering phases

3.1 Requirements Analysis

The main goal of this phase is to formalize the user needs detailed in a requirements analysis document using UML.

This phase is composed of the following activities:

1. **Activity A:** Actors and external entities definition, carried out by use case diagrams called *Actors General View* (1) and *External Entities General View* (2);
2. **Activity B:** Definition of the passive context (collaboration between the external *entities* and the modeled system), carried out by interaction diagrams called *External Entities Dynamic Data Flow* (3) and *External Entities Static Data Flow* (4);
3. **Activity C:** This is decomposed in three steps:
 - Definition of Use Cases, performed by:
 - defining the use cases and adding them into the Use Case General View (5),
 - describing the use cases (collaboration between the Actors and the modeled system) using interaction, activities, and class diagrams (8).
 - Creation of a *Data Flow General View* (6) diagram that shows all the interactions between the system and all the actors and external entities.
 - Domain Analysis, carried out by at least a class diagram called *Domain Classes* (7).

3.2 Object Analysis

The main goal of this phase is to produce the first logical organization (package organization) of the classes found during the first phase and to complete these classes by adding attributes, inheritance, *etc.*

This phase is composed by **Activity D:** Produce for each Use Case of a class diagram called by the Use Case name, *i.e.*: *FirstUseCase* (8). This diagram contains all the classes identified in the sequence diagrams that describe the Use Case. Organise these classes into a set of packages. For each package, create a class diagram called by the package name, *i.e.*: "*ThirdDesignPackage*" (9).

Note that, if necessary, create activity and state diagrams for the classes' behavior modeling.

3.3 Architectural Design

The main goal of this phase is to produce the first architectural view of the modeled system. In this phase, the analysis packages become logical components. **Note:** *The first logical organization could be changed here if the*

designers consider that the logical organization does not respect the design constraints.

This phase is composed by the next activities:

1. **Activity E:** The dependencies between the identified packages are presented in the *Package General View* (10) diagram, then the packages' interface classes are identified and the packages collaboration is presented in the *Package Collaboration General View* diagram (11).
2. **Activity F:** For each package, add or modify classes (by addition of methods or attributes) (9). If necessary, add activity or state diagrams describing the collaboration between these classes.
3. **Activities G and H:** Modify the package diagrams by the addition of design components or the application of design patterns.

3.4 Object Design

The main goal of this phase (for each package) is to produce the detailed architectural organization (complete classes, attributes, methods, diagrams, etc.; find new classes, new packages, etc.) to achieve the package description started in the activity F.

This phase is composed of the following activities:

1. **Activity I:** Upgrade the contents of each existing diagram.
2. **Activity J:** Create the man-machine interface packages and description diagrams.
3. **Activity K:** Create the database interface packages and description diagrams.

3.5 Physical Design

The main goal of this phase is to produce the physical architecture. For that, one has to design nodes and processes, and describe their collaboration.

This phase is composed of the following activities:

1. **Activity α :** Creation of hardware component (nodes and devices) deployment diagram (12).
2. **Activity β :** Processes identification and their affectation to the nodes of the deployment diagram. Identification of the collaboration between the processes in a diagram called *Process Collaboration General View* (13).
3. **Activity γ :** Identification of components (from our design or from existing frame works) and their affectation of them to the processes in the *Process General View* diagram (14). Allocation of the classes to the components (one or more classes by component).

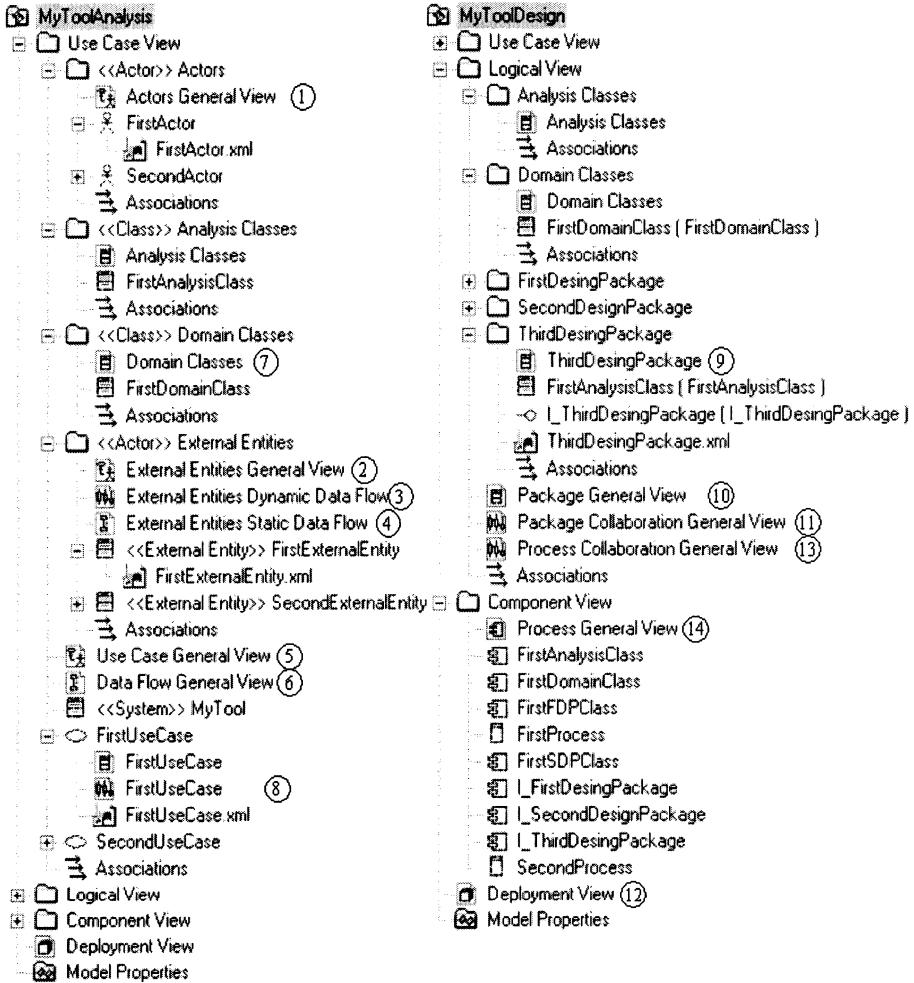


Figure 3. View of the instrumented browser structure

4. NEPTUNE TOOLS: DESCRIPTION AND USAGE

We present here the Neptune tools as well as their characteristics and their usage.

4.1 The Checker

The UML notation includes a constraint language (OCL) whose purpose is to express properties over a model that cannot be specified graphically. OCL is mainly based on first-order logic and model navigation expressions. Constraints expressed in OCL can be used in different contexts: within a

class, they express class invariants; within an operation, they express pre- and post-conditions. OCL expressions are also used to specify guards of transitions. The upcoming version 2.0 of UML proposes an extension of OCL contexts to actions, therefore allowing the specification of properties over behavioral features: it is possible to declare which messages can be sent within a given operation. The NEPTUNE project has also made its own proposals of extension, namely the capability to compute transitive closures of navigation expressions and to specify temporal properties of dynamic diagrams.

The verification of OCL constraints is in general not possible statically during the design and the analysis. For example, a class invariant can only be checked at run-time when exiting from an operation of the class. Consequently, even if this technique is not widely available now, OCL constraints can only be translated into Boolean expressions and inserted into the code generated from the user model.

However, UML class diagrams can also be used as a meta-language (i.e. a natural or formal language used to describe another language) to define the abstract syntax of the UML notation itself. Meta-class invariants expressed as OCL formulas must now be verified on UML models, which are made of the finite set of modeling elements, contained in the UML diagrams. Verifying that a model satisfies an OCL constraint becomes now decidable and this is the purpose of the NEPTUNE checker.

Consequently, the purpose of the checking tool developed in NEPTUNE is twofold:

- Verify the well-formedness of OCL constraints introduced at the application level;
- Check that an UML model satisfies the OCL constraints defined at the meta-level.

This second point enlightens another objective of NEPTUNE. The ability to check meta-level constraints over application level models makes it possible to check UML well-formedness rules as well as methodological rules as soon as they can be expressed as constraints over the UML meta-model.

4.1.1 Tool Description

As described in Figure 4, the OCL checker takes as input a database of OCL rules, together with the UML metamodel and a user model. Both models are stored in XMI files whose DTDs are compatible with version 1.4 of UML. It returns diagnostic information indicating which rules are syntactically or semantically incorrect, and the model element on which rules fail. As most current tools only export UML 1.3, some transformation must be applied to their output XMI so that it becomes compatible with UML 1.4. This transformation is performed using an XSLT processor. XMI files are then

parsed and transformed into an internal form that is accessed by the checker through an API compatible with UML 1.4.

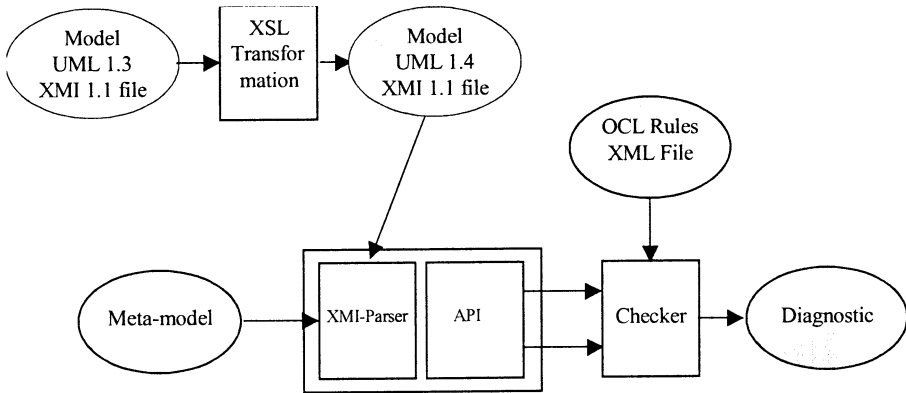


Figure 4. Checking process

The checker loads the metamodel and uses it to validate and compile the OCL rules. The model is then loaded and traversed so that each model element is made an instance of its associated meta-class.

4.1.2 Checking Rules

This section presents a classification of the OCL constraints verified by the NEPTUNE checker. As previously mentioned, they can be split into two subsets: rules attached to meta-level classes and constraining the structure of every UML model and, rules included in a user model and constraining the implementation of the model.

4.1.2.1 Meta-model Level Rules

The starting point for the specification of this set of rules is the UML semantics document. It uses the OCL language to formalize constraints over meta-model instances. We have enriched this first set of well-formedness rules by including inter-diagram coherence rules, software engineering rules, target language rules, and business process rules, i.e. a description of a set of related activities that, when correctly performed, will satisfy an explicit business goal.

- **Well-formedness rules:** Most of the rules, appearing in the UML semantics reference document, are intra-diagram rules. They add structural constraints over model diagrams that cannot be expressed graphically at the meta-model level.
- **Inter-diagram coherence rules:** Inter-diagram rules require checking properties between at least two different kinds of diagrams. They are in

general not specified in the UML semantics document and may depend on the UML methodology. Inter-diagram rules include type-checking rules that require verification of method calls encapsulated in messages relative to the class contents and the inheritance hierarchy. At the object level, the number of links between objects in a collaboration diagram must conform to the multiplicity declaration of the corresponding associations. Sequence diagrams describing possible execution traces can also be checked against state-transition diagrams describing all possible execution paths.

- **Target language oriented rules:** Target language rules are rules that require checking properties dependant on the target language chosen for the implementation. For example, one can check the Java requirement disallowing multiple inheritance of classes. Such rules must be grouped in some package and conditionally checked.
- **Software engineering rules:** They address the verification of properties that depend on the methodological process imposed to develop an application and on metric constraints defining modeling rules. The first point concerns the definition of rules applicable at the end of each phase of a development. They are defined in accordance with the NEPTUNE process and concern, for example, the way that external actors must be specified and their interaction with the system. The second point is more project dependent and can express naming conventions, size limitations for classes or packages, *etc.*
- **Business process rules:** Business process methodology is well integrated into the software engineering methodology and thus requirements concerning design and analysis phases are mostly similar. The business process design relies on methodological rules, which describe the mapping of business concepts into the UML notation. The E-P business extensions have a stereotyped note for defining rules. The note is stereotyped <<business rule>> and is attached with a dashed line to the model element (class, operation, etc.) to which it applies. Three categories of business rules are extracted:
 - Derivation rules define how information in one form may be transformed into another form, or how to derive some information from another piece of information.
 - Constraints govern the structure and the behaviour of objects or processes, *i.e.*, the way objects are related to each other or the way object or process changes may occur.
 - Existence governs when a specific object may exist – usually inherent in the class model.

4.1.2.2 Model Level Rules

Model level rules are defined by the application designer and are associated to user-level model elements in order to constrain the application

code. The model element to which a rule applies is called its context and can be a class, a method, or a dynamic diagram. The different kind of rules are:

- **Class invariant and operations pre/post conditions:** This usage of OCL constraints is already allowed by UML. The NEPTUNE tool only checks their syntax and semantics. A more extensive static checking would require proof techniques;
- **Action clauses proposed by UML/OCL 2.0:** They offer the capability to specify which messages can be sent by an operation, thus allowing some kind of verifications over dynamic diagrams. However, such verification is limited to a one step transition. The current version of NEPTUNE checker only verifies the semantics' correctness of these rules;
- **Temporal constraints over dynamic diagrams:** The NEPTUNE tool introduces the ability to specify temporal properties over dynamic diagrams through an extension of the OCL language. Temporal operators are borrowed from the Computational Tree Logic (CTL) and applied to sequence and state/transition diagrams. In the first case, properties over the occurrences of specific messages are asserted. In the second case, properties concern the state of the current object.

4.1.3 Tool Usage

The NEPTUNE user has to define the set of model elements and the set of OCL rules he/she wants to apply to the model elements.

Model elements are selected either from the meta-model browser or from the model browser. In the first case, the NEPTUNE user can select all the classes of the model while in the second case he/she can select all the classes of a given package of the model.

The NEPTUNE user selects rules by choosing a subset of the rules database identified by a keyword. These subsets correspond to the rules categories presented in Section 4.1.2 and to the different phases of the NEPTUNE methodology. Such a subset can be further restricted selecting rules individually. Different subsets can be merged to define a new subset that can be named and reused later. The user can also define his own set of rules using the rule editor.

Once the set of model elements and the set of rules have been determined, the checker is launched. It produces a diagnostic showing, for each violated rule, the model elements that do not respect it. To help the navigation within this result file, paths leading to failing model elements are enlightened in the model and meta-model browsers.

Figure 5 summarizes the use of the NEPTUNE checker.

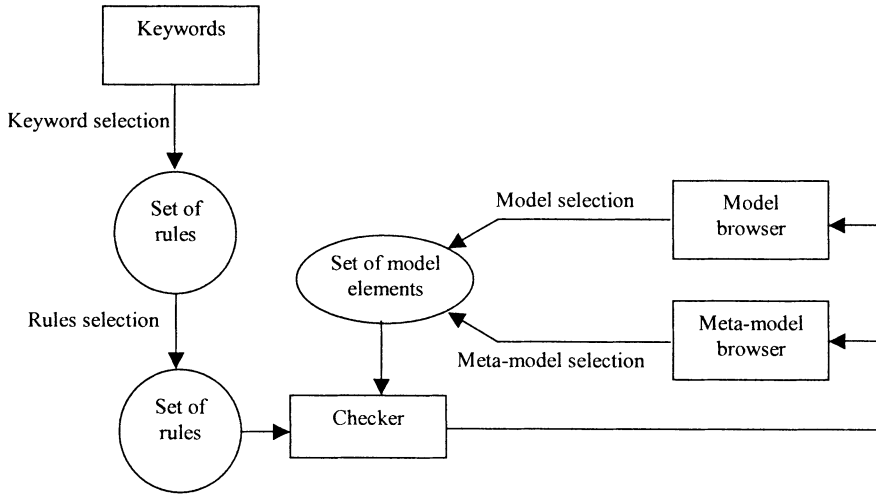


Figure 5. NEPTUNE checker general use

4.2 The Document Generator

We provide here a description of the document generator and how to use it.

4.2.1 Tool Description

With the emergence of UML as a standard in software development, software projects are now very often based on this technology. This statement is true in many business fields, like software engineering of course, but also for knowledge management or business process. Although all the participants in a software-engineering project may be able to read and understand an UML model, this does not necessarily happen with people in other domains. Thus, it is useful to transform the information contained in a UML model into different representations, easier to read for someone not necessarily being fully skilled in UML.

The main objective of the document generator is thus the production of professional documentation that results from the exploitation of UML model atomic parts, to whatever business field they are related. This documentation is an end-user-oriented documentation, which takes into account the professional expertise of the reader.

The principle of generation is based on a model-correlated interpretation of documentary elements. A documentary element specifies the type of

information to extract from the model and the kind of transformation to be applied on it.

Each documentary element will be processed to generate a small part of the final document.

Grouped together, they build up a template, called shape in the NEPTUNE context.

For each business field, we have identified a set of standard documents. An example is the validation plan for software engineering. The tool provides one shape for each of these typical documents. Each of them can be applied on a UML model, in order to generate the appropriate document.

NEPTUNE also takes into consideration the specificity of each project. Thus, the documentation designer can create new shapes, totally independent from the ones aforementioned, fully dedicated to the UML model to be documented.

Moreover, if the set of transformations available in NEPTUNE does not exactly fit the user needs, they can write their own transformations.

An additional feature of the generator consists in its ability to deal with external documentation. In other words, the NEPTUNE user can add to a shape several references to external documentation. Of course, the only transformations handling these external pieces of documentation are the ones that have been designed to do so.

The technologies involved in the process are XMI for the UML model, XML for the documentation, and XSL for the transformation aspects.

4.2.2 Tool Usage

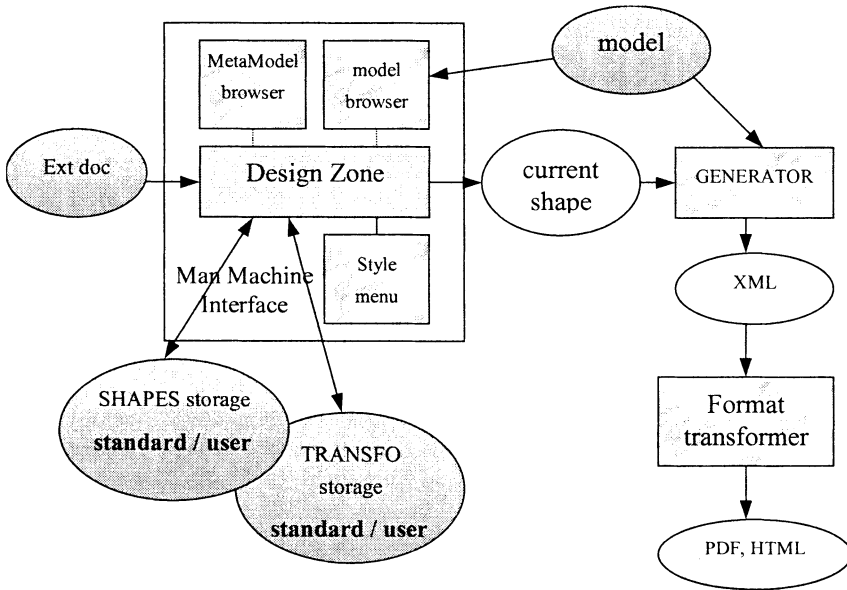


Figure 6. The document generator usage

The NEPTUNE user has first to define the aspect of the documentation they want to produce. This is done through the selection of a standard shape, or through the creation of a new one, called user shape.

The shape is built up in the design zone and can be stored at any time. A shape is composed of both structure and contents information. Regarding the rendering of the structure (titles, sub-titles), a style menu offers a selection of fonts, polices, sizes, etc.

Concerning the contents, each paragraph of the final documentation is the output of a transformation performed on a part of the UML model (diagram, class, etc). Each atomic part of the model can be selected with the model browser. For systematic treatment (over all classes for example), the documentation designer can use the UML browser in which all UML notions appear.

The transformations aim to turn into textual, business dedicated and user-friendly form, a specific semantic information extracted from the model. Once the shape is selected or customized, the next step consists in feeding the generator with both this shape and the UML model to be documented. The output of the generator is the final document in XML format.

The last feature of NEPTUNE document generator is the format transformer. Through choice of the documentation designer, this feature is

able to turn the XML document previously produced into another format. It can be either RTF, PDF, or HTML.

4.2.3 Documentation Shapes and Transformations

In order to describe the final documentation, the user designs a shape, which contains two kinds of information:

- the hierarchical structure of the document,
- the contents of the document.

4.2.3.1 Hierarchical Structure

The structure of the shape is tree-based. Leaves of this tree represent the contents of the documentation, while nodes describe its organization. In the final document, the nodes will be the titles of the chapters and subchapters, while the leaves will contain the text, the tables and all kind of information the user wants to show. While creating a shape, the user can add new descendant nodes to any already existing node. Figure 7 shows an example of a simple shape structure and its result in the documentation.

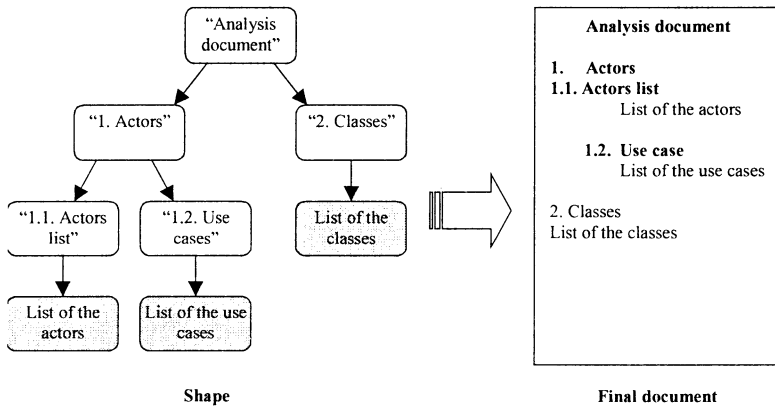


Figure 7. Example of shape structure

4.2.3.2 Content

As said above, the leaves of a shape contain the information needed to generate the corresponding part of the final document. Each leaf is a documentary element that consists of:

- an XSL transformation;
- some UML model element targets;
- links to XML external documentation.

NEPTUNE extracts information from an XMI-format UML model by using XSL transformations. The final document part associated with a leaf is obtained by applying the transformation on the XMI model, using model element targets and external documentation.

For instance, a transformation could, given an actor, retrieve all the use cases associated with this actor.

NEPTUNE will come with a set of predefined transformations, but the user can design his own transformations, tailored to his needs.

4.2.4 Example

For the software engineering domain, NEPTUNE can generate three specific documents from a UML model:

- the Design Analysis document that converts the UML Model into a document understandable by anyone,
- the Validation Plan based on the model, that defines the validation tests that should be performed to test the software application.
- the Interface Manual, that describes the interfaces of the modeled software with much more accuracy than needed for the Design Analysis document.

In this context, we show how we generate a validation plan from a UML model with the NEPTUNE tool.

First, we have to specify that UML/NEPTUNE process recommends that each use case should be documented in a textual form, respecting the following categories:

Table 1. Use case description

Summary	Brief presentation of a use case
Use context	Conditions of use by triggering elements (frequency of activation, synchronous or asynchronous triggering, etc.)
Triggering element	Actor or use case
Pre-conditions	Stable system condition necessary before use case can be accomplished
Input data	Data used
Description	Detailed description of interaction between triggering elements and the system
Post-conditions	Stable condition achieved by the system at the end of the interaction with a use case
Output data	Data produced
Exceptions	Error condition that the system cannot resolve

Next, for us a "validation plan" is composed of "validation cases" that contain "test cases". A "validation case" corresponds to one "use case" from the UML model. A "test case" is a sequence of scenarios. An "execution case" is an execution for a particular scenario. So there is at least as many



"validation cases" as "use cases". For each one, there is at least a set of "tests sequences". Each test is described in a "Test Description Form":

Table 2. Text description form

Text Description Form 1	
Project: Project Name	Software version:
Test ident:	Test title: Use Case name
Test type: Functional Robustness Performances Ergonomic Documentation	
OBJECTIVES: Use context field of the Use Case External Documentation	
PRE-REQUIREMENTS: Pre-conditions field of the Use Case External Documentation	
PROGRESS INSTRUCTIONS: Description field of the Use Case External Documentation	
INPUT: Input data field of the Use Case External Documentation	
EXPECTED RESULTS: Exceptions field of the Use Case External Documentation	
OBSERVED PROCESS:	
OBSERVED RESULTS:	

This "Test Description Form" is generated from the UML model (in XMI format) and the XML files attached to the UML model. This approach allows us to have an automatic generation of the validation plan.

5. CONCLUSION AND PERSPECTIVES

The main contribution of the project is to make UML easier to use. The NEPTUNE process provides a "ready to use" instantiation of the Unified Process with guidelines for most of the problems a user faces when they use UML for the first time or in a new context. This process has been built from the "know-how" acquired from intensive UML usage on development of many projects. Furthermore, the NEPTUNE tools allows the user to make reliable models by checking the set of rules associated with the process and allowing the user to extend these rules by defining their own set.

The NEPTUNE tools also provide ways to document the model with default templates and transformations that can be extended or customized as needed.

As the documentation is generated from a model, it can always be up-to-date: after each model improvement, the associated documentation can be generated again without changes.

The user can define document shapes, transformations and checking rules so that the NEPTUNE tools can be extended and customized as needed.

All the NEPTUNE tools use as entries XMI and XML and are totally independent from the case tools used to design the UML model.

The purpose of the NEPTUNE consortium is also to extend the use of UML and the NEPTUNE tools to different domains in which it will be useful to model processes or organizations using UML.

ACKNOWLEDGMENTS

This collaborative work would not been achieved without the active contribution of Pierre Bazex, Louis Feraud, Christophe Lecamus, Ralph Sobek, Christian Percebois, Yannick Cassaing, Antoine Jammes, Laurent Pomies and Etienne Roblet.

Chapter 7

The OOSPICE Assessment Component: Customizing Software Process Assessment to CBD

Friedrich Stallinger, Brian Henderson-Sellers and John Torgersson
Kepler University Linz, Austria; University of Technology, Sydney, Australia; University of Borås, Sweden

Abstract: OOSPICE is a collaborative EU project which is developing a CBD capability assessment approach along the lines of ISO 15504 together with a development methodology for CBD consistent with this assessment approach. There are several elements to this, one of which is the creation of a process assessment methodology specific to the needs of CBD. This chapter focuses on this assessment methodology component of the overall OOSPICE project. Based on the description of the underlying CBD Process Reference Model forming the common basis for the assessment as well as the CBD methodology, we provide details on the project's CBD assessment model and show how this model is implemented in the assessment tool and how the application of these two elements to real world CBD projects is supported through an assessment method.

Key words: Software process assessment, software process improvement, component-based software engineering

1. BACKGROUND TO THE OOSPICE PROJECT

According to many analysts, a key to overcoming the bottleneck in software productivity can be found in increasing the reuse of components, i.e. to assemble systems from more-or-less finished building blocks. Since the mid nineties, the software engineering community has been witnessing the growing popularity of this component-based development (CBD) approach which is radically changing the way in which systems are analysed,

architected, implemented, transitioned and evolved. However, the simple idea of CBD is hard to achieve [1] and companies are experiencing many problems in adopting or applying CBD; for instance, organisational structures and processes that are not adapted to CBD, inappropriate technical and management approaches that are being used for CBD, a general lack of knowledge and skills to obtain the benefits of CBD regarding quality and productivity, deficient or inadequate information concerning availability, quality and reliability of software components as well as capability of component suppliers and their component construction process, or a prevailing culture that is resistant to CBD, rejecting newness and mistrusting components.

The demands of rapidly changing, in particular internet-based, business environments [2] have led to solution providers devising or adopting a variety of ways to design and build software systems from third-party as well as their own components. At the CBD implementation level, component infrastructure technologies such as the OMG's CORBA (Common Object Request Broker Architecture), Sun's JavaBeans and Enterprise JavaBeans, and Microsoft's (Distributed) Component Object Model (DCOM/COM) have become widely accepted de facto industry standards; yet methods and processes supporting CBD are still not commonplace. While various suitable processes for CBD are currently still emerging and their general applicability is still being debated, systematic and generally accepted approaches to the standardization of such development processes for CBD are still missing. Like other development approaches CBD would benefit greatly from the availability of repeatable processes for building, selecting and assembling modules – here components.

The fields of software process assessment (SPA) and software process improvement (SPI) target very similar goals to those of CBD - improvements in time-to-market, cost and quality. Nevertheless, established SPI approaches such as CMMI and SPICE [3] are not ideally suited to either object-oriented development or CBD, constituting a significant gap in technology. They generally lack tailoring and customisation to CBD, in particular with respect to terminology or adequacy and granularity of the underlying process or assessment models. Furthermore, the relationship between CBD and the way in which components themselves are produced is often not clear. As there are no widely accepted processes specifically for CBD, this creates problems on applying SPA (and SPI) to component-based development.

The EU-funded international research and development project OOSPICE (Object-Oriented Software Process Improvement and Capability dEtermination for Object-Oriented and Component Based Software Development, IST-1999-29073) aims to overcome the above shortcomings with respect to standardisation and assessment and improvement of processes for CBD. Based on the principles of empirical software engineering it

combines the four major concepts of CBD, object-oriented development, software process assessment and software process improvement. It focuses on the processes, technology and quality in software development using component-based development.

Its main technological objectives are the evaluation of current theory and practice in CBD, the development of a unified CBD process model with reference to best practice and underpinning process metamodel, the development of a CBD assessment methodology consisting of an assessment model, an assessment method and an assessment software tool, the development of a CBD methodology for 'architecting' and assembly of independently produced components and for component provision together with a CBD software tool, and the definition of capability profiles for component providers through the analysis of results of the CBD process assessment methodology. There is a further goal of ensuring that the results of the project gain both international acceptance and take-up by industry through dissemination, standards and licence schemes. In particular, an extension to the ISO 15504 process assessment standard will be proposed. In achieving these project objectives, the task of major initial importance is the definition of a unified CBD process model able to serve two functions: as a basis for a methodology for component-based development (see Chapter 8 for details), and also as a process reference model and source of process definition for capability assessment.

The underlying process reference model for CBD and its relation to the assessment model as well as an overview on the initial assessment methodology for CBD are described in the remainder of the chapter. Section 2 provides some background on software process assessment and software process improvement and outlines the differences between traditional and component-based software engineering, while Section 3 discusses the underlying process reference model for CBD. Section 4 describes the CBD assessment methodology, including a description of the structure of the assessment model and an overview of the assessment method and tool. Conclusions and insights gained so far and the outlook on further work round up the chapter.

2. SOFTWARE PROCESS ASSESSMENT AND IMPROVEMENT AND CBD

A promising approach for tackling the problem that software development is generally too late, too costly and results in products of insufficient quality is to ensure that software is developed via a well-defined high-quality process. To reach this goal and to continuously improve the software development process, it is necessary to regularly assess its efficiency and capability and

decide on and implement improvements. This approach is called Software Process Assessment and Improvement, with Software Process Improvement (SPI) intending to provide a comprehensive set of industry best-practice processes that a software-developing organization should have implemented together with a framework specifying how to continuously improve these processes.

Although the assessment of process capability is a relatively new technique in the software industry with formal approaches dating back to the pioneering work of Humphrey [4], a number of assessment-based methods for SPI has been developed and results from empirical research, e.g. studies by the Software Engineering Institute [5] or data from the SPICE Trials, have demonstrated the efficacy of process improvement approaches based upon the results of process assessment and shown that process improvement efforts can lead to gains in productivity, time to market and quality, that in total add up in returns on invested resources.

The emergence of a number of competing approaches to process assessment in the early '90s finally led to the establishment of the SPICE Project and the development of an international standard, ISO 15504 [6], intended to harmonize the various (generally traditional) software engineering-oriented approaches to software process assessment.

On the other hand, CBD has a major impact on traditional software engineering processes, mainly through the split of the overall development activities into multiple parallel *component provisioning* and *application assembly* tracks and the fundamental separation and independence between *specification* and *implementation* and the basic side effects resulting from these, such as the focus on architectural tasks, the need for integration of multiple component provisioning strategies (e.g. in-house development of components, acquisition of components, wrapping of legacy systems or asset integration in general), the need for specific project planning tasks together with the complex dependencies that occur between project management and the component management in general.

Hence, CBD raises questions about whether traditional software engineering approaches are appropriate in this emerging field and, as a consequence, whether existing approaches to software process assessment and improvement are appropriate for the assessment and improvement of CBD. In general, CBD is *integration-centric*, emphasising the selection, acquisition, and integration of components from in-house sources and/or external vendors, with the latter including the use of commercial off-the-shelf (COTS) products or open source components. Furthermore, companies gaining competitive advantage by acquiring third-party components, which may thus be either 'black box' or 'white box', often have to develop their own components when third-party components do not fully satisfy system requirements [7] – contributing to significant variation in CBD processes and complicating these

processes as well as their assessment. Indeed, findings from software process assessments indicate that some approaches to software process assessment have difficulties in evaluating process capability for CBD. These studies tend to confirm expert views that the practices in CBD in many of its forms are not always adequately mapped to the process models used in assessment approaches. The customizing of software process assessment approaches to CBD has been almost non-existent although available evidence indicates that there is a distinct need.

Hence the OOSPICE project focuses on the processes employed for both the provision and development of reusable software components and for the architecting and assembly of larger systems from components, with the goal of overcoming the above (and future) shortcomings (e.g. inadequate terminology, inadequate granularity of processes) –that may be experienced when applying current assessment approaches to CBD by developing a process reference model for CBD and an assessment methodology based on this process reference model. This approach is favoured by the evolution of ISO 15504 to a generic framework for process assessment [8] applicable across a wide range of domains in the course of its transition from a technical report to an international standard. Under the resulting revised architecture of ISO 15504, specific process models may be established for different domains, with the status of the models ensured through appropriate standardisation mechanisms.

3. CREATING A PROCESS REFERENCE MODEL FOR CBD

The Unified CBD Process Model under development by OOSPICE plays a central role within the project. It is defined as a *set of process component specifications* describing the software life cycle processes for CBD. Its main purpose is to provide the common basis for the assessment methodology work, in particular the assessment model and the underlying Process Reference Model (PRM), and the CBD methodology work (Chapter 8) and to ensure consistency of these two development tracks. The process model contains *processes* and *tasks* that are to be applied during software development and maintenance and describes the involved *work products*. The scope of the model covers the development of software applications based on the use of software components as well as for developing components that are independently deployable. Attention is also paid to converting legacy software into components.

Table 1. OOSPICE CBD processes by process categories [9]

Customer-Supplier	Support
Acquisition Preparation	Maintenance

Supplier Selection Supplier Monitoring Customer Acceptance Supply Customer Support Operational Use	Configuration Management Documentation Problem Resolution Joint Review Verification Validation Product Evaluation Quality Assurance Audit Usability
Modelling	Management
Domain Engineering Business Modelling Requirements Engineering Behaviour Specification Architecture Provisioning Strategy User Interface Specification	Programme Management Project Management Risk Management Measurement Quality management Infrastructure
Application Assembly	Organisation
Application Internal Design Component Assembling Application Testing Application Delivery Component Selection	Process Establishment Process Assessment Process Improvement Asset Management Reuse Programme Management
Component Provisioning	Human Resources
Component Internal Design Component Testing Component Delivery Legacy Mining	Human Resource Management Training Knowledge Management

Compared to traditional software engineering, CBD has several major aspects that are reflected in the CBD process model as developed by OOSPICE (cf. also Section 2): firstly, its starting point is a clear understanding of the business model representing the context in which a solution has to be provided and acting as a major input for architectural activities; secondly, CBD is a methodology based on behaviour analysis with roots in the behaviour-based Catalysis methodology developed by D'Souza and Wills [10]; thirdly, the behaviour is assigned to component specifications that are used in a component specification architecture; finally a key aspect of CBD is to decide strategically upon the provisioning of each component specification, i.e. acquisition from an external source, wrapping of a legacy system, adaptation of an existing component, in-house development, etc. As a consequence, the traditional engineering processes have to be split into a multiple track development process consisting of an *application assembly track* focusing on solving a business problem and a *component provisioning track* focusing on delivering a single component which can be used as a software building block in one or more applications. The resulting OOSPICE CBD engineering processes are shown in Table 1 grouped into the process categories *Modelling*, *Application Assembly* and *Component Provisioning*

together with the processes of the non-engineering process categories *Support, Customer-Supplier, Management, Organisation and Human Resources*.

The process components of the OOSPICE process model are defined in terms of *Name; Purpose; Outline; Tasks; Inputs and Outputs* and do not yet fully describe the internal details of the processes, e.g. techniques or aspects of sequencing. The resulting CBD Process Model on the one hand forms the starting point for the definition of the CBD Methodology in which these process details are fleshed out, and, on the other hand, for deriving a Process Reference Model (PRM) as the basis for the assessment methodology work. Further details on the underlying architecture of the OOSPICE models and on the CBD methodology can be found in Chapter 8.

The process reference model is the starting point for the OOSPICE capability assessment model. A PRM is a model that defines the processes that are to be assessed by providing basic definitions of these processes. It is defined in terms of *Name; Purpose* and *Outcomes* only and is thus derivable by identifying *Outcomes* from the *Tasks*, together with *Inputs* and *Outputs* of the process model. An Assessment Model (cf. Section 4.1) is then created to be compatible with the PRM by adding indicators of capability to a subset of processes of the PRM that are used to assess the capability of the respective processes. The resulting capability profile evaluates the maturity and success of the implemented processes in terms of these quality indicators. The assessment model thus draws on the PRM and embeds the measurement framework of ISO 15504 [11].

One of the aims of the OOSPICE project is to propose the PRM for CBD as an extension to the ISO/IEC 15504 process assessment standard and thus to extend the capabilities of ISO15504 to support CBD.

4. THE CBD ASSESSMENT METHODOLOGY

The OOSPICE CBD Assessment Methodology provides the framework for assessing software processes in a component-based environment. It aims to be used in all kinds of organisations and market sectors that have adopted a component-based approach to software development.

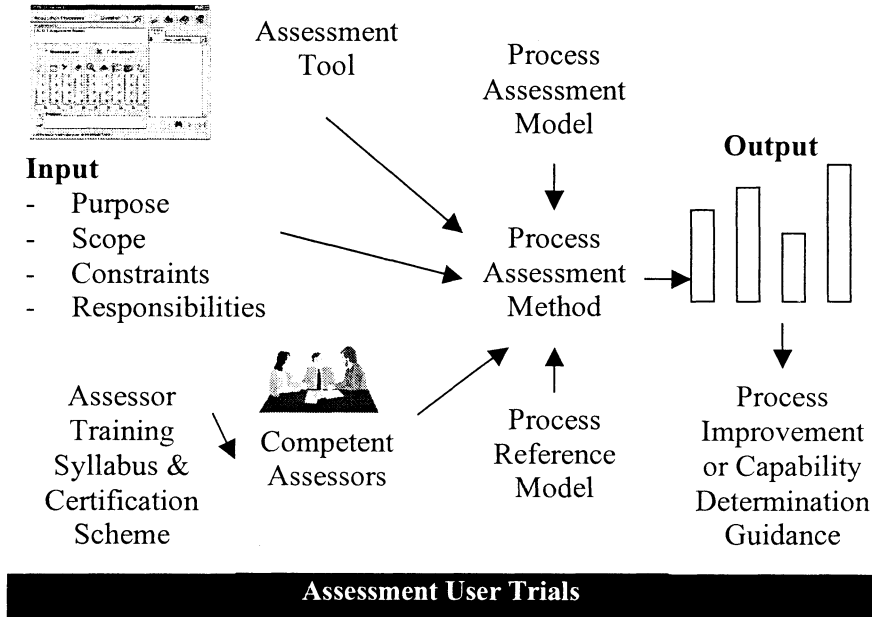


Figure 1. Overview on the OOSPICE assessment methodology components

Figure 1 provides an overview on the single components of the assessment methodology and how they work together. At the core of the methodology is the Assessment Method (see Section 4.2) that defines how an assessment shall be performed within the assessment framework, i.e. how the assessment input is transformed into the assessment output (mainly capability profiles and assessment findings) that are further used for process improvement or capability determination. The means used for measuring the capability of the assessed project or organisation is the Process Assessment Model (PAM) (see Section 4.1) that incorporates the underlying Process Reference Model (PRM) containing the processes for component-based software development. An Assessment Tool (see Section 4.3) is then used to aid collection of assessment data, rating and presentation of the results. An assessment is normally led by a Competent Assessor who has the required knowledge, skills and experience. The OOSPICE project will further define the basis for the operation of a certification scheme for assessors together with an assessor training syllabus for training courses.

Furthermore, the main components of the OOSPICE Assessment Methodology, namely the assessment method, the assessment model and the assessment tool, are subject to validation through Assessment User Trials.

The main components of the Assessment Methodology are described in the following subsections.

4.1 CBD Assessment Model

The transition to a component based approach for software development among a large number of organisations has led to a new way of performing software development, using different processes and a new terminology. Therefore a need for processes, suitable in a CBD environment, and a way to assess their capability has increased.

The current international standard for software process assessment, ISO/IEC 15504, has proven to be suitable for general software development approaches. However, there is a lack of coverage of CBD-specific issues both when it comes to the content within the processes and in the terminology used. Thus, there is a gap and the aim of the OOSPICE assessment methodology model is to bridge this gap.

The PRM described in Section 3 defines a process framework to be used when developing software with a component-based approach. The assessment model, described in this chapter, elaborates on the CBD Process Reference Model (PRM) in order to facilitate capability assessment of these processes.

4.1.1 Structure of the Model

Like the PRM, the assessment model is a two-dimensional model containing a process dimension and a capability dimension. The relationship between the PRM and assessment model is a one-to-one relationship between the process categories, processes, purpose statements, process capability levels and the process attributes. However, the PRM doesn't contain a sufficient level of detail to be used alone as the basis for reliable assessments of process capability. Therefore, the assessment model needs to provide a comprehensive set of indicators of process performance and capability.

Indicators work as guidance when assessing the different processes. Figure 2 shows the structure of the indicators used within the assessment model. The indicators are divided into process performance and process capability. To indicate process performance, for each process, a set of base practices is provided along with the input and output work products and their characteristics. The base practices are a description of the unique activities of the process and the work products represent the input and outputs from these activities. This is where the major CBD characteristics can be seen. The indicators of process performance are closely related to the process outcomes defined in the PRM and address the achievement of the process purpose.

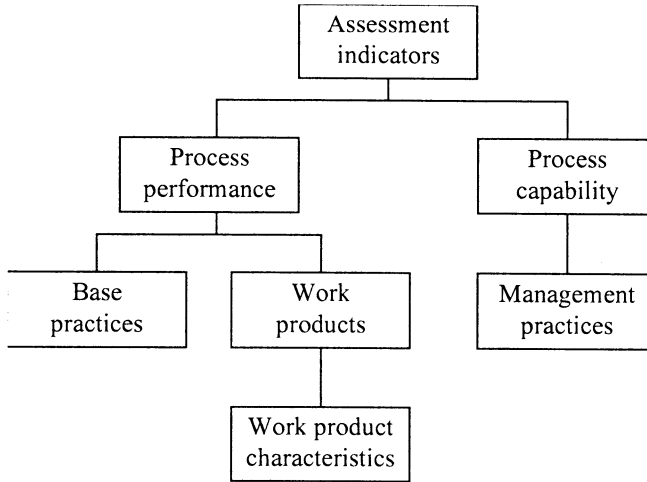


Figure 2. Structure of assessment indicators

Management practices are provided as indicators for process capability and are the means of achieving the capabilities, addressed by process attributes. Evidence of management practice performance supports the judgement of the degree of achievement of the process attribute.

The indicators of process capability are applicable to all processes in the PRM whereas the indicators of process performance are associated with one particular process.

4.1.2 Field of Application

The purpose of the PAM is to provide an assessment model suitable for assessments of all organisations that are using a CBD approach – not only those that have implemented CBD to its full extent but also those organisations that are planning to adapt CBD in order to find out in what direction the organisation should move to get closer to a full implementation of CBD and how to improve their process capability.

The assessment model can be used as an internal tool with the purpose of improving the organisation's processes as well as a tool to verify a component supplier.

4.2 CBD Assessment Method

In order to perform consistent and repeatable ratings of process capability, the current international standard for software process assessment, ISO/IEC 15504, defines a set of requirements for the performance of an assessment.

The CBD assessment method is developed to be conformant to the requirements in the standard.

The assessment method together with the assessment model represents the core element of the CBD Assessment Methodology. The objective of the method is to provide as much help and guidance as possible for the assessment team in order to facilitate the assessment process and improve the quality and reliability of the assessment result. Therefore, the method emphasizes the use of detailed templates that work as outputs from the assessment and are associated with one or several of the tasks that are performed within the method. This makes the output from different assessments comparable and decreases the likelihood of leaving out important information when reporting the result back to the assessed organisation.

4.2.1 Structure of the Method

The method is grouped into three phases: preparation-, assessment and reporting phase. Each phase contains a purpose and a set of activities. The purpose states the overall objectives of the phase and gives an indication about what to accomplish within the phase. Each phase is further divided into activities. Each activity has a general description and a number of tasks.

The description is in short terms what is to be done within the activity. The tasks are a further elaboration about how to meet the requirements of the activity and the responsibility for carrying out the tasks. The roles needed to carry out the activity conform to the tasks as well as the output(s) produced. The outputs are in form of templates, which are documents that help the assessors to collect the data and present the result(s).

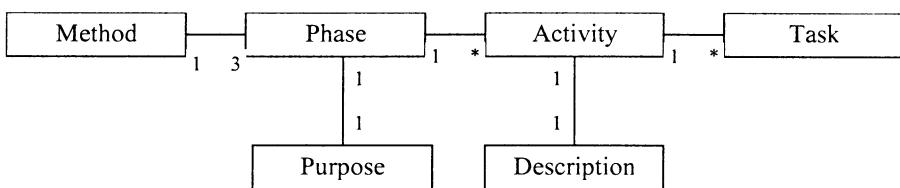


Figure 3. Structure of the CBD assessment method

In the method, a set of guidelines is also provided and is incorporated as assistance in using the assessment method. The guidelines serve as additional information about how to conduct the different activities.

4.2.2 Performing the Assessment

As mentioned in section 4.2.1 the method is divided into a preparation, assessment and reporting phase. These are phases that an assessment team will go through during any assessment. Figure 4 provides an overview of the phases and activities that are described in the method. The rounded rectangles show the phases which have a set of activities connected to them.

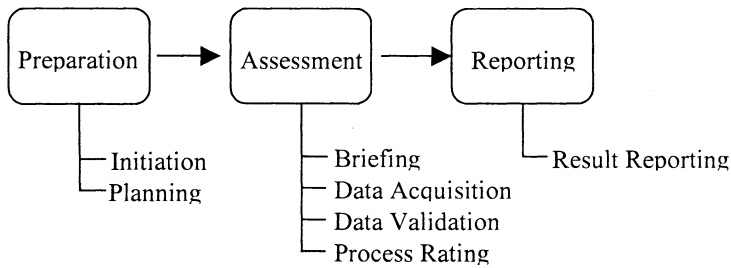


Figure 4. Phases and activities in the assessment method

4.2.2.1 Preparation Phase

During the preparation phase, the organisation to be assessed and the assessment team prepare for the upcoming assessment phase. The major part of this phase is the planning and scheduling of the assessment. The assessment plan is created containing information about the assessment scope, roles and responsibilities and the key activities to be performed during the assessment.

4.2.2.2 Assessment Phase

Most of the activities during the Assessment Phase are performed on site within the assessed organisation. In this phase the assessment team and the participants from the assessed organisation are briefed on the purpose with the assessment as well as the methodology to be used. After that, interviews are performed to collect data sufficient to meet the assessment purpose and scope and to rate each process attribute up to and including the highest capability level defined in the assessment scope. The rating is based on objective evidence gathered from the collection and validation of the required data.

4.2.2.3 Reporting Phase

When the data have been collected and the different processes rated, the result is reported back to the assessed organisation. The report will highlight

the process profiles, key results, perceived strengths and weaknesses, identified risk factors and potential improvement actions.

4.3 CBD Assessment Tool

An Assessment Tool is being developed within the OOSPICE Project to aid collection of data during an assessment, rating of process attributes and presentation of the assessment results to the assessed organisation.

The assessment tool implements the assessment model and provides easy, generally hyperlinked, access to the elements of the assessment model and their definitions. Figure 5 shows an example screen for data collection and process attribute rating. Figure 6 provides a sample presentation of assessment results.

The tool is currently developed to the stage of an internal prototype for usage in and support of the project's Assessment User Trials. The final version will be available for download at www.oospice.com.

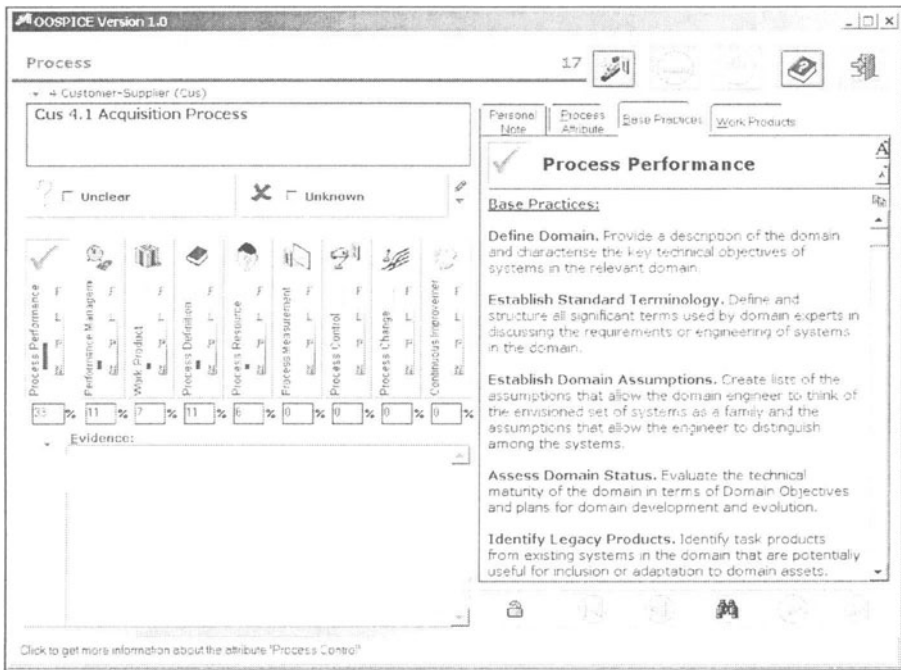


Figure 5. Sample screen of the OOSPICE Assessment Tool for assessment data collection and process attribute rating

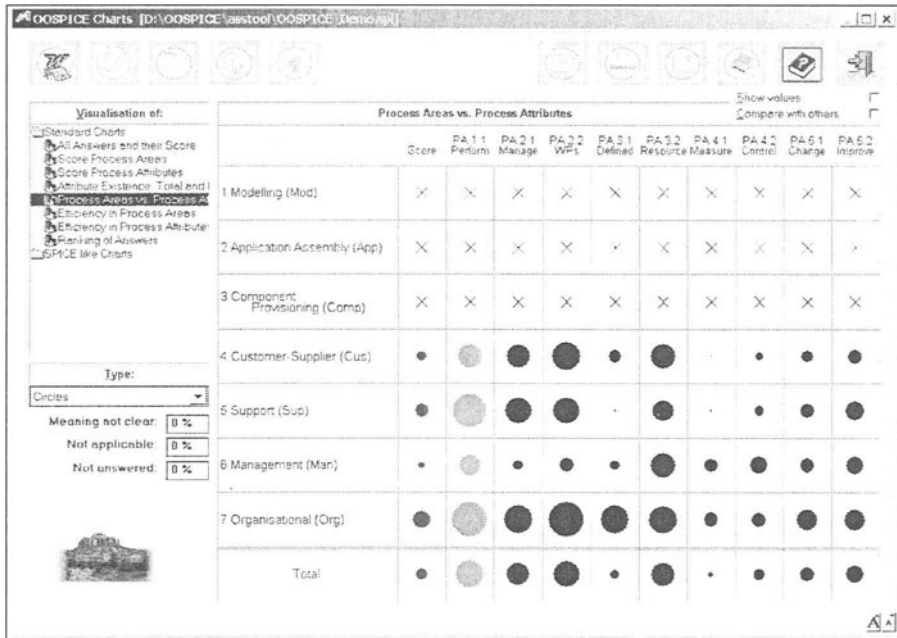


Figure 6. Sample chart of the OOSPICE Assessment Tool for assessment result presentation

5. CONCLUSION

The international research and development project OOSPICE aims to overcome the gaps and shortcomings experienced when applying current software process assessment and improvement approaches to CBD and to provide a CBD methodology (see Chapter 8) consistent with the developed process assessment and improvement approach. It is based on component-based software engineering principles and encompasses processes and process engineering, on the one hand, and process capability assessment on the other. In this chapter we have focused on the CBD process model developed by OOSPICE that is forming the common basis for both development strands. We have shown how an ISO 15504 conformant process reference model can be derived from this process model and how this process reference model forms the basis of a CBD assessment methodology consisting of an assessment model, method and tool as main elements. These methodology components are currently available as initial, project internal versions and subject to ongoing work and, in particular, about to undergo validation within Assessment User Trials. The final results will be available as soon as possible. It is also intended to propose the final results, in particular the developed

process reference model, for extension of the ISO 15504 process assessment standard to CBD.

ACKNOWLEDGEMENTS

The OOSPICE Project is developed as EU-funded shared-cost RTD project under contract number IST-1999-29073 within the European Commission's Fifth Framework Programme.

The OOSPICE project partners are:

- Institute of Systems Sciences, Department of Systems Engineering and Automation, Kepler University Linz (Austria) (Coordinator)
- Computer Associates (Belgium)
- WAVE Solutions Information Technology GmbH (Austria)
- Huber Computer Datenverarbeitung GmbH (Austria)
- University of Boras (Sweden)
- Volvo Information Technology (Sweden)
- Griffith University (Australia)
- Centre for Object Technology and Application Research (Australia)

This is Contribution Number 02/03 of the Centre for Object Technology Applications and Research (COTAR).

The introductory sections of this paper are based in part on material published in Stallinger *et al.* [12].

We also wish to thank the Australian Research Council for funding the Australian contribution to this project.

REFERENCES

- [1] Blechar, M., Gartner Group Report: *Chasing the Elusive Productivity of Components*, November 1999.
- [2] Feldman S., *The Objects of E-Commerce*, OOPSLA '99 keynote address, Denver, November 1999.
- [3] ISO/IEC TR 15504: 1998 – *Software process assessment*. ISO 1998.
- [4] Humphrey, W.S. and Sweet W.L., *A Method for Assessing the Software Engineering Capability of Contractors*, Report CMU/SEI-87-TR-23, Software Engineering Institute, Pittsburgh, September 1987.
- [5] Herbsleb, J., A. Carleton, J. Rozum, J. Siegel and D. Zubrow, *Benefits of CMM Based Software Process Improvement: Initial Results*, Report CMU/SEI-94-TR-13, Software Engineering Institute, Pittsburgh, August 1994.
- [6] Rout, T.P., The SPICE Project: Past, Present and Future, *Software Process '96*, Brighton, December 1996.

- [7] Seacord, R. C., *Custom vs. Off-The-Shelf Architecture*, Technical Note CMU/SEI-99-TN-006, 1999.
- [8] ISO/IEC JTC1/SC7, High Level Design for the Revision of ISO/IEC TR 15504:1998, Document SC7 N2210, Oct. 1999.
- [9] OOSPICE Partners: D5.1 – Initial CBD Process Model, OOSPICE Internal Deliverable, OOSPICE Partners, May 2002.
- [10] D’Souza, D.F., Wills, A.C., 1999, *Objects, Components and Frameworks with UML. The Catalysis Approach*, Addison-Wesley, Reading, MA, USA, 785pp.
- [11] ISO/IEC 15504, 2002, International Standards Organization (in press)
- [12] Stallinger, F., et al.: *Software Process Improvement for Component-based Software Engineering: an Introduction to the OOSPICE project*. Procs. Euromicro 2002 Conference, IEEE Computer Society Press, Los Alamitos, CA, USA.

Chapter 8

The OOSPICE Methodology Component: Creating a CBD Process Standard

Brian Henderson-Sellers, Friedrich Stallinger and Bruno Lefever
University of Technology, Sydney, Australia; Kepler University Linz, Austria; Computer Associates, Belgium

Abstract: OOSPICE is a collaborative EU project that is developing a CBD capability assessment approach along the lines of ISO15504. There are several elements to this, one of which is the creation of a new methodology and underpinning metamodel for CBD. This chapter focuses on these latter two components of the overall OOSPICE project. We will describe both the genesis and the results of the process components, their metamodel descriptions and their realization in the construction of a project-specific CBD process. We will also show how these elements are linked to and provide the basis for the Process Reference Model, Assessment Method and Assessment Model.

Key words: OOSPICE, capability assessment, CBD methodology

1. BACKGROUND TO OOSPICE PROJECT

Component-Based Development (CBD) is a new style of software engineering, which has been emerging in industry since the mid nineties. Driven by the pressure of industry to deliver solutions in a fast and efficient way, it resulted largely from a reaction to the software engineering activities of the early nineties where many development shops concentrated on rewriting their portfolio of solutions from scratch in a more flexible architectural approach. Many of these initiatives stalled when developers and managers began to realize that it took a relatively long time to re-develop an existing core system that had been developed, enhanced and used for, for example, more than fifteen years. The core questions remained: firstly, why

one should re-write software that already fulfils the requirements and, secondly, whether the fact that this software is written using traditional, old fashioned underlying technology is a sufficient reason in itself for this re-writing.

CBD is a development approach capable of creating new systems from blends of old systems, brand new developments, acquired packages or components in a mix of different technologies. It shifts the focus of software development from core in-house development to a development process focussing on the use of internally or externally supplied components, as well as promising increased efficiency and flexibility in development as well as of the developed products. These resulting products are no longer products with some kind of 'closed nature' but, instead, component-based products that can be integrated with other software products.

While the increase in the popularity of CBD has largely been driven by the appearance of standards or quasi-standards on the product side (e.g. component models such as COM/DCOM, JavaBeans, CORBA), the community is lacking systematic and widely accepted approaches to standardized development processes for CBD. Indeed, the use of component-based development has a major impact on traditional software engineering processes. This results mainly from the split of the overall development activities into *component provisioning* and *application assembly* tracks and the basic side effects resulting from this distinction, such as:

- the fundamental independence between specification and implementation,
- the focus on architectural tasks,
- the need for integration of multiple component provisioning strategies, such as inhouse development of components, acquisition of external components, wrapping of legacy systems or asset integration in general,
- the necessity of a central component catalogue and mechanisms to foster the exchange of components, and
- the need for specific project planning tasks, as well as complex dependencies between project management and the component management in general.

In addition, the components need, themselves, to be created. Thus *component architecting* also needs to be supported. These three aspects of CBD (component architecting, provisioning and assembly) suggest the need for the introduction of new elements into existing process element descriptions such as those found in ISO12207, ISO15504, the OPEN Process Framework [1], Catalysis [2] and similar documents.

On the other hand, the fields of Software Process Assessment (SPA) and Software Process Improvement (SPI) share very similar goals of CBD – shorter time-to-market, reduced costs and increased quality. They provide a wide spectrum of approaches to the evaluation and improvement of software processes (CMM, CMMI, SPICE etc.) with considerable advances in the

standardization of these approaches [3] as well as of the underlying process models (e.g. [4]), but generally lack tailoring and customization to CBD.

The EU-funded international research and development project OOSPICE (OOSPICE is an acronym for **S**oftware **P**rocess **I**mprovement and **C**apability **d**etermination for **O**bject-**O**riented/**C**omponent-based Software Development, IST-1999-29073) builds on these developments within process assessment standardisation. To overcome the shortcomings experienced when applying current assessment approaches to CBD (e.g. inadequate terminology, inadequate granularity of processes), OOSPICE focuses on the processes, technology and quality in software development using component-based development. Based on the principles of empirical software engineering it combines four major concepts: CBD, object-oriented development, software process assessment and software process improvement. Its main objectives are a unified CBD process model and underpinning metamodel, a CBD assessment methodology consisting of an assessment model, an assessment method and an assessment software tool, and a CBD methodology together with a CBD software tool.

The first work done in the OOSPICE project towards creating process/methodological support created an interim set of process component specifications – a Process Model that is relatively summary in nature and forms the springboard for the Assessment Model and encompasses the Process Reference Model (PRM) as well as being the “bare bones” of the CBD Methodology to be developed later. Major inputs to the project were specified as the various ISO standards, especially 15504 and 12207, and the OPEN Process Framework [5], [1].

The CBD Process Model, which is the focus of this chapter, plays a central role in the project by ensuring consistency of both the assessment methodology and the CBD methodology parts of the project. The Process Reference Model (PRM) for CBD derivable from this Process Model is intended to be proposed as an extension to the ISO/IEC 15504 process assessment standard. The underlying work on harmonizing terminology and on establishing the architecture necessary for the various process and assessment models of the OOSPICE project and an overview on the initial Process Model as well as an outlook on how they will be further used in the project are described in the remainder of the chapter. Section 2 describes the necessary OOSPICE terminology, while Section 3 describes the specific methodological requirements of CBD. Section 4 discusses the creation of the CBD methodology, including a description of the architecture of the various process and assessment models of the OOSPICE project and how they relate to each other. Section 5 provides an overview on the initial proposals for the CBD process metamodel. Conclusions and insights gained so far and the outlook on further work round up the chapter.

2. TERMINOLOGY FOR PROCESS REFERENCE MODEL, ASSESSMENT MODEL, METHODOLOGY AND EXISTING ISO STANDARDS

A major challenge to the development of the various OOSPICE models was the differing terminology across the various subfields of process being united. The key issues relate to the understanding of the key term “process”. These differences derive primarily from the different heritages of the two main approaches being merged. Within the standards arena, that is ISO 12207 and ISO 15504, the approach derives from the process management principles embedded in concepts of Total Quality Management [6], [7]. Within this framework, driven largely by the success of the Capability Maturity Model [8], the goal for effective improvement of organisational maturity is to establish a “defined process” for each instantiation; the role of the process was to transform defined inputs into outputs, and a “defined process” was seen as achieving this in a repeatable and measurable manner. In defining such a process, a key element is the establishment of work components (activities or tasks) that made up the process; thus, the process is seen as a series of elements assembled to achieve the transformation of inputs to outputs. Hence, we obtain the following definitions for process:

“A sequence of steps performed for a given purpose; for example, the software development process.” [9] – also used in the CMM for Software [8].

“A system of operation or series of actions, changes, or functions, that bring about an end or result including the transition criteria for progressing from one stage or process step to the next.” [10] – also used in the Systems Engineering CMM.

ISO 12207 [4] introduces the term “activity” as an identifier for the “steps” or “actions” referenced in earlier standards: “A set of interrelated activities, which transform inputs into outputs.” This definition became the generally accepted standard, and is employed in EIA IS 731 [11] and ISO 15504 [3].

ISO 12207 also introduces the hierarchical decomposition of processes – a process is composed of a set of activities, and each activity in turn is seen as comprising one or more tasks. The reason for this derives from the need to differentiate between “normative” and “informative” elements; in the context of the standard, the activities were elements that are required to be performed, the tasks were elements that could support achievement of the activities. The use of a series of directions or prescriptions for the performance of the process leads to the use of the term “process implementation model” – a model

constructed in this way provides instructions for the implementation or execution of the process.

With the introduction of concepts of process assessment, the terminology can be further confused. The purpose of process assessment is to evaluate the capability of the processes under evaluation, either directly or through the evaluation of “organizational maturity” as with the CMM. “Process capability”, however, is a term that has an accepted understanding in the area of statistical process control and reflects the probability of achieving desired results from the execution of a process. Within the specific domain of process assessment, the emphasis on statistical control disappears and the term is defined as “the range of expected results that can be achieved by following a process” (CMM for Software) or, less rigorously, as “the ability of a process to achieve a required goal” [3]. Process capability is evaluated by determining the extent to which the process achieves its purpose efficiently and effectively. In order to achieve this, the process is defined, not in terms of the tasks and activities required to implement it, but in terms of its purpose and the outcomes of its implementation. Thus, ISO 15504, while retaining the definition of process from ISO 12207, defines processes in terms of purpose and outcome statements. The new amendment to ISO 12207, now in the final stages of approval, adopts this approach to process definition, while maintaining the specification of tasks and activities for process execution.

An important aspect of process assessment is the acceptance of the concept that a process need not be formally defined or documented in order to achieve its purpose. An informally performed process is seen as capable of generating the specified outcomes, although not in any planned or controlled manner. The establishment of a documented, and then a defined, process is seen as representing the achievement of significant capability.

This concept of the “informally performed process” establishes a key point of potential confusion between the terminology in the standards domain and that employed in the work of the Object Management Group, and the OPEN Consortium, in deriving de facto standards for modeling languages (UML) and processes (SPEM). In this context, a process is seen as a documented series of actions; the formal definition of process from ISO 12207 and related standards equates to “activity” in the OPEN framework. The concept of the “informally performed process” is not found within OPEN [5].

Resolution of these various terms and their appropriate meta-level attribution is one of the aims of this chapter. A terminology mapping is given in Table 1. In the following sections, we will generally use OOSPICE terminology (last column in table), sometimes together with the OPEN name in brackets.

In conclusion, then, it can be seen that the main difference in terminology between OPEN and ISO standards is the definition of the word process: In OPEN the process is the “whole process” (OOSPICE: methodology) of an organization or project. In contrast, the ISO standards define process as components supporting the product life-cycle. In ISO 12207, a process consists of activities and activities consist of tasks. In OOSPICE, we skip the activity level. Thus, processes consist directly of tasks. Henderson-Sellers *et al.* [12] conclude that for OOSPICE a process can be defined simply by:

A process is a set of interrelated tasks, which transform inputs into outputs.

A process consists directly of tasks; but what was missing was an OOSPICE terminological equivalent of OPEN’s use of the word process to mean the “whole process” of an organization or project. The term “methodology” is adopted for use in OOSPICE to describe the “process” at the highest, full lifecycle scale.

Table 1. Terminology mapping between ISO standards, OPEN and OOSPICE (after [12])

Description of term	Name in 15504	Name in 12207	Name in OPEN	Name in OOSPICE
Totality often exemplified by its documentation (external and internal)	Documented process	Software Life Cycle Process	Process	Methodology
Collection of definitions of process components, with full documentation on (external) specification	Process Dimension	N/A	Set of Process Components	Process Model
Something transforming inputs to outputs. Also, an individual, documented instance of an M2 level definition	Process	Process	Activity (a kind of Process Component)	Process
Mid-range conversion of inputs to outputs	N/A	Activity	Task	Task
Smaller scale conversion of inputs to outputs	N/A	Task	No special name – essentially steps within the Task description	
Full process in enactment	N/A	N/A	Process Instance	Implemented Processes

3. SPECIFIC METHODOLOGICAL REQUIREMENTS RELATED TO COMPONENTS

Setting up a methodology for component-based development assumes a thorough understanding of the basic concept: the component itself. Components have been given many different definitions in the literature. Absolutely essential is the explicit distinction between the component specification and the component implementation. Components provides the required flexibility: to be able to change the implementation aspects without any consumer of the component's services noticing this change. This makes the component implementation of no relevance to its consumer as long as the behavior does not change. This means that, as a first step in the approach, only component specifications are important, in accordance with the much-lauded principle of information hiding [13].

It is exactly this focus on behavior as an explicit modeling focus that underlines the difference between traditional software engineering and the component-based one. Modeling behavior as an explicit concept is quite new to most software engineers, although advocated in some OO methodologies. Behavior-based modeling forms the cornerstone of a good CBD approach. Instead of concentrating on classes and their behavior, you concentrate on the behavior first and derive the classes afterwards.

Behavior of components is captured in interfaces. From the identification of these interfaces the component specifications can be derived. This is achieved by applying grouping techniques like clustering.

Behavior specification architecture is the most important engineering process identified in OOSPICE concentrating on the identification and definition of the component specifications supporting a particular aspect of the business. This architecture will define the overall interactions between different components and will ensure that their collaboration fulfils the required functionality. Having defined this behavior specification architecture, the signal for two disciplines is given: the process of realizing the defined behavior, the component provisioning, and the process of consuming component specifications into end-user applications, the application assembly. Each of these component-based development subtracks requires specific skills. Component provisioning will, most of the time, deal with persistent storage of information and implementation of the core business rules while application assembly will focus on the user interface and workflow management aspects.

A component specification may have different possibilities regarding implementation. The process to make a cost efficient choice for the implementation of a component specification identifies the need for a good

provisioning strategy in order to support a cost efficient approach to component implementation. Finally, in OOSPICE, it is the component management process that keeps track of all component specifications and underpins a reuse strategy.

4. CREATING A CBD METHODOLOGY FROM THE PROCESS MODEL

The underlying architecture for OOSPICE is based on that of the Object Management Group, as exemplified through its use for defining the Unified Modeling Language [14]. In OOSPICE, we use the lowest three levels of the multi-level OMG/UML architecture and apply this to realm of process (Figure 1). The lower level relates to processes as implemented and assessed (on a single project); the middle or M1 level is the “process as published”. This might mean an organization’s process handbook. This is the (full lifecycle) process that is often labelled by the vendor or supplier with a well-known name or “branding” and should provide *everything* necessary for software development. This process is defined by a set of rules at the uppermost or M2 level – this is the process metamodel.

In OOSPICE, this architecture is refined (Figure 2), particularly at the M1 level. At this abstraction level are a process model (or set of process components) together with the M2-level metamodel. The former will be a major focus of this chapter and described in more detail in Section 5. The Process Model has recently been baselined and work progresses on the Methodology towards delivering the techniques, together with construction guidelines, needed for its creation.

Using the construction guidelines, a process engineer or methodologist can create a methodology for a specific business domain, a specific organization or indeed for a specific project. At the organization level, this is the methodology that is used repeatedly by that organization as “their standard methodology/process” when undertaking CBD. The methodology, also at level M1, adds sequencing and configuration information and constraints to the process components. It provides a fully documented approach for building component-based systems: the OOSPICE “methodology”.

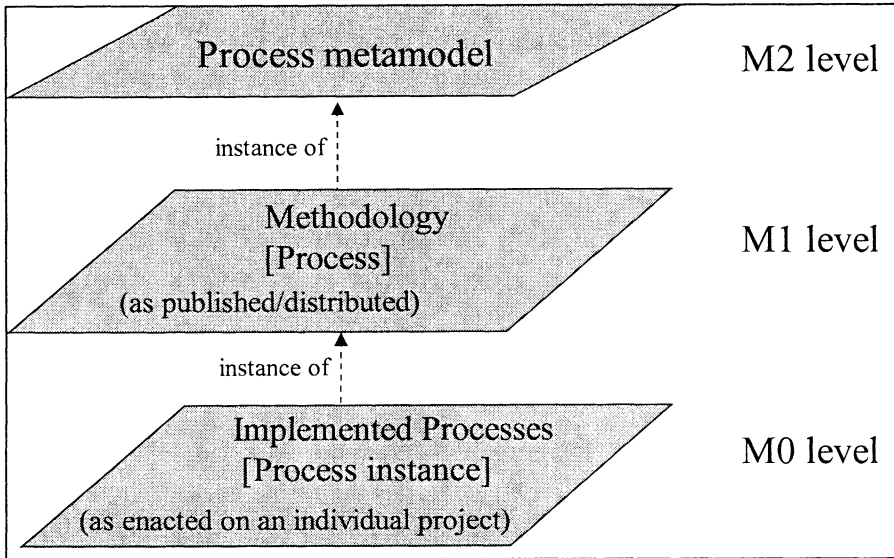


Figure 1. The lowest three levels of the process metamodel architecture. OOSPICE terminology is used. The OPEN terminology is shown in square brackets (after [12])

As well as the methodology aspect of OOSPICE, there is the capability assessment support. Originating from a different cultural viewpoint, assessment is applied at the M0 level of the implemented process. An assessment model, based on and compatible with a Process Reference Model (PRM), is used for this purpose (Figure 3). To some degree, the assessment side mirrors the methodology side, as seen in this figure, in which there are two major links: a common “Implemented Processes” element and a complementarity between the tasks of the process construction framework and the differently orientated purpose and outcomes of the assessment modeling component of the framework.

We should also note that the architecture of Figure 3 is technology-independent. Although it has been derived in the OOSPICE project in the context of CBD, nowhere is component technology critical to, or even influential upon, the derived architecture. What *is* technology-dependent is the detailed-level content of the models and the process elements in the repository.

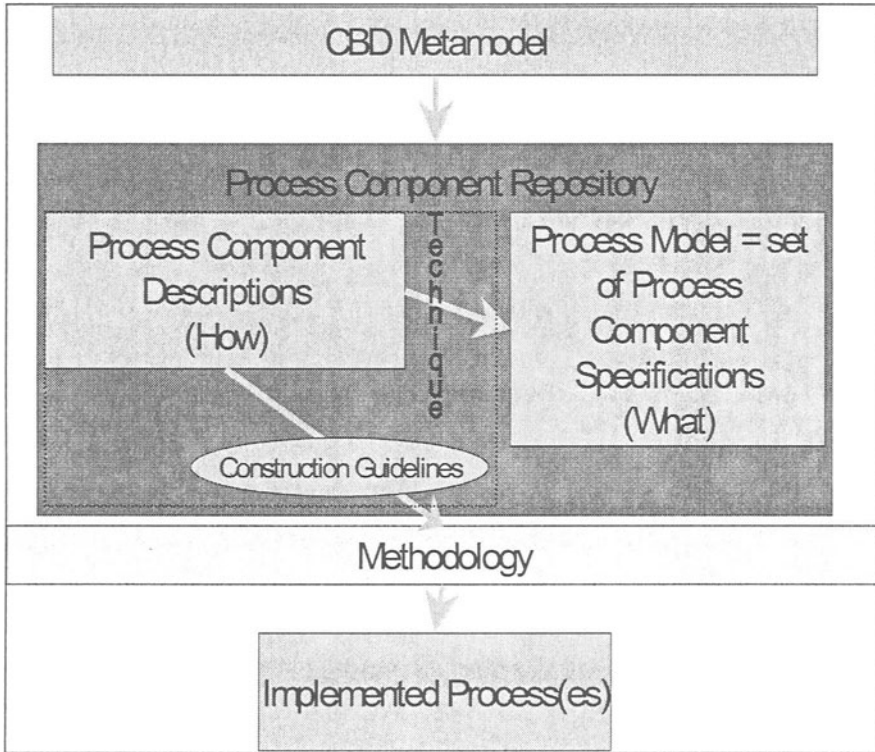


Figure 2. Details of the M2, M1 and M0 levels of the process construction components of the overall OOSPICE architecture (modified from [12])

Torgersson and Dorling [15] note that the process model describes and structures the type of processes to be used when using a CBD approach for software development. At a high level, two groupings were identified (primarily to apportion work): processes which could be described as “engineering processes” (for the development and assembly of components) and processes which define the non-engineering processes i.e. those generally needed to support and place the engineering processes in an organisational context. These process sets have been further subdivided by classifying all the processes into 42 groupings (see Table 1 of Chapter 7), although this grouping should be regarded as a first draft and subject to change.

Although the form of these process component specifications is unique to OOSPICE, their origins are mostly from best practice and from other documents, particularly the OPEN process components [16] and the two aforementioned ISO standards and their supplements. For each process component definition required in OOSPICE, we first inspect the two ISO standards [3], [4] or the OPF Repository [1] to see if such a definition already exists. If it does, as is often the case, then we reuse that (see [12]). We then

identify areas *not* covered in these standards, which are all the specifications related to the emerging discipline of CBD. These are then written from scratch based on existing industry best practice and published theoretical and empirical research, especially from Catalysis [2] and industry experience of Computer Associates and colleagues. The result is a document that is comprehensive in its coverage of all CBD-related process component specifications but does not attempt to fully describe the internal details – to be addressed later in the development of the OOSPICE Methodology. This is then translated into purpose and outcomes and augmented as necessary to provide a compatible metamodel for the process assessment component of OOSPICE.

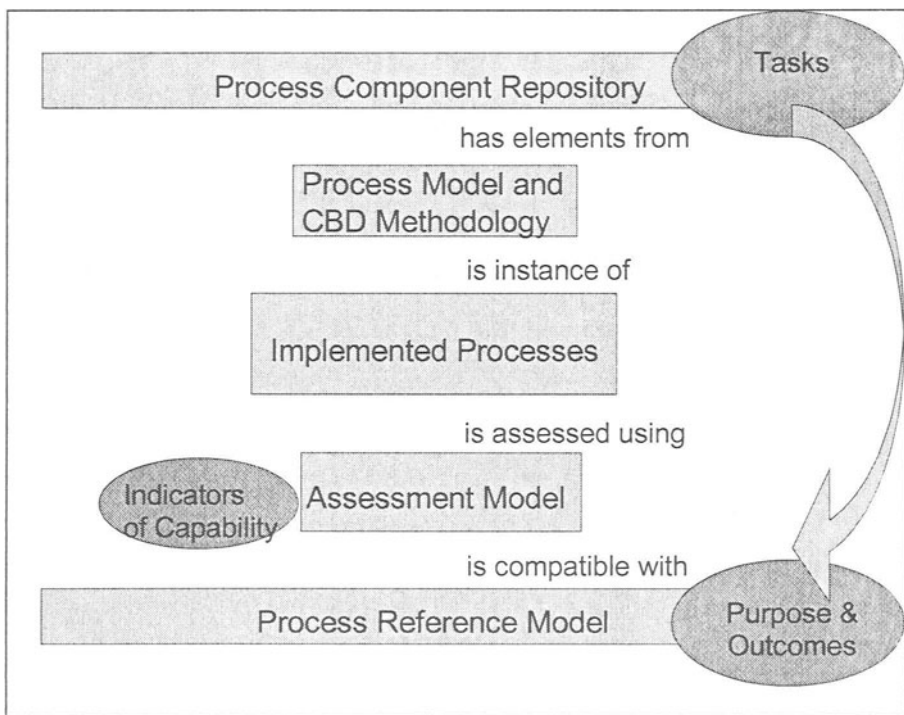


Figure 3. The overall architecture of the M1 level of OOSPICE (modified from [12])

The CBD Process Model is defined in terms of Name; Purpose; Outcomes and Tasks. A Process Reference Model (PRM), on the other hand, is defined only in terms of Name; Purpose and Outcomes (e.g. [15]). Given the Process Model, the PRM is thus readily derivable by omitting the Tasks.

It is the PRM that is the starting point for the OOSPICE capability assessment model (Figure 3). A process reference model is one that defines those processes that are to be assessed by providing basic definitions of these

processes. An Assessment Model is then created to be compatible with the PRM by adding indicators of capability to a subset of processes of the PRM. These indicators are used in calculating the capability profile for each process in the assessment model. The capability profile evaluates the maturity and success of the implemented process in terms of these quality indicators. An assessment model thus draws from the process reference model and embeds the measurement framework of, say, ISO 15504 [17]. The result is known as a capability rating and the data created by applying the assessment procedure to the enacted project are known as the “capability profile” of that organization. For further details see Chapter 7.

On the other hand, it is the Process Model, not the Process Reference Model, that is the basis for the development of the methodology (see Figure 2). To create this, each of the Task outlines of the Process Model will be examined and, as necessary, fleshed out. More importantly, the Techniques appropriate to all these Tasks will be elucidated, again from best practice, including documented techniques in the OO and CBD literature. However, fleshed out Tasks and Techniques still lack the sequencing necessary for process enactment. The final task of OOSPICE’s methodology development component will be to investigate appropriate process sequences and also to create a set of construction guidelines that will permit the process engineer to fabricate appropriate processes which extend those that the OOSPICE methodology development team will provide as exemplars.

5. THE UNDERPINNING METAMODEL

Within the three layer architecture of Figure 1, it can be seen that the process components are all instantiated from a single metamodel (at the M2 level). Previous SPI approaches have not included such a metamodel for process or capability assessment. One original aspect of the OOSPICE project is that it will follow the lead taken by OMG and CDIF in formally identifying an appropriate metamodel (at the OMG M2 level) to create a formal consolidation for the process model and process capability assessment.

The methodological aspects can be well modeled by a typical OO process metamodel. Here we use the OPEN Process Framework (OPF)’s metamodel as a starting point. In addition, a number of additional metatypes related to Work Products specific to CBD have been identified and an initial draft [18] has been distributed internally to the OOSPICE team members. Together, this work and the OPF, supplemented if necessary by concepts from the OMG’s SPEM architecture [19], are being used as the basic input to the creation of the underpinning metamodel for the methodological aspects of the OOSPICE project. Finally, capability assessment concepts need to be added to this evolving metamodel – something not previously attempted.

When finalized, the metamodel, process model, reference model, capability assessment model and methodology will be offered to ISO for international standardization.

6. CONCLUSION

The OOSPICE project is based on Component-Based Software Engineering principles. It encompasses processes and process engineering, on the one hand, and process capability assessment on the other. Bringing together these two subdisciplines of software development, in this chapter we have outlined the necessary mappings between differing terminologies and detailed the OOSPICE model architecture in which we are developing not only a CBD methodology and underpinning metamodel and capability assessment model and methodology (see Chapter 7) but also a process reference model (PRM) and a CBD process model (which encompasses much of the PRM). When the descriptions of the processes are expanded into purpose, outline and tasks they form the process model whereas the subset using only purpose and outcomes forms the basis for the assessment strand of OOSPICE. Further development of this strand as well as most of the CBD methodology strand are the topic of ongoing work. It is intended that the final results will be offered to ISO for standardization.

ACKNOWLEDGEMENTS

This is Contribution Number 02/04 of the Centre for Object Technology Applications and Research.

The OOSPICE Project is developed as EU-funded shared-cost RTD project under contract number IST-1999-29073 within the European Commission's Fifth Framework Programme.

The OOSPICE project partners are:

- Institute of Systems Sciences, Department of Systems Engineering and Automation, Kepler University Linz (Austria) (Coordinator)
- Computer Associates (Belgium)
- WAVE Solutions Information Technology GmbH (Austria)
- Huber Computer Datenverarbeitung GmbH (Austria)
- University of Boras (Sweden)
- Volvo Information Technology (Sweden)
- Griffith University (Australia)
- Centre for Object Technology and Application Research (Australia)

We also wish to thank the Australian Research Council for funding the Australian contribution to this project.

This chapter is based in part on material published in Henderson-Sellers *et al.* [12], [20], for which we also thank Jörn Bohling and Terry Rout. We also acknowledge the valuable work contributed to the Process Model by members of Middlesex University, UK. We also wish to thank Cesar Gonzalez-Perez and Tom McBride for their helpful comments on an earlier draft of this chapter.

REFERENCES

- [1] Firesmith, D.G. and Henderson-Sellers, B., 2002, *The OPEN Process Framework. An Introduction*, Addison-Wesley, Harlow, UK
- [2] D'Souza, D.F., Wills, A.C., 1999, Objects, Components and Frameworks with UML. The Catalysis Approach, Addison-Wesley, Reading, MA, USA, 785pp
- [3] ISO/IEC TR15504, 1998, Information Technology – software process assessment, in 9 parts, International Standards Organization
- [4] ISO/IEC 12207, 1995, Information Technology – software lifecycle processes, International Standards Organization
- [5] Graham, I., Henderson-Sellers, B. and Younessi, H., 1997, *The OPEN Process Specification*, Addison-Wesley, UK, 314pp
- [6] Deming, W.E., 1982, *Out of the Crisis*, MIT Center for Advanced Engineering Study, Cambridge, MA, USA
- [7] Crosby P.B., 1979, *Quality is Free : the art of making quality certain*, New American Library, New York, NY, USA
- [8] Paulk, M.C., Weber, C.V., Garcia, S., Chrissis, M.B. and Bush, M., 1993, *Key Practices of the Capability Maturity Model, Version 1.1*, CMU/SEI-93-TR-25, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA (February 1993)
- [9] IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology
- [10] IEEE Std 1220-1998, IEEE Standard for Application and Management of the Systems Engineering Process
- [11] Electronic Industries Association. Systems Engineering Capability Model (EIA/IS-731), 1998, Electronic Industries Association, Washington, D.C., USA
- [12] Henderson-Sellers, B., Bohling, J. and Rout, T., 2002a, Creating the OOSPICE model architecture – a case of reuse, *Procs. SPICE 2002, Venice, Palazzo Papafava, March 13-15, 2002* (ed. T. Rout), Qualital, Italy, 171-181
- [13] Parnas, D., 1972, On the criteria to be used in decomposing systems into modules, *Communications of the ACM*, 15(2), 1053-1058
- [14] OMG, 2001a, OMG Unified Modeling Language Specification, Version 1.4, September 2001, OMG document formal/01-09-68 through 80 (13 documents) [Online]. Available from: <http://www.omg.org>
- [15] Torgersson, J. and Dorling, A., 2002, OOSPICE – the road to qualitative CBD, *Procs. SPICE 2002, Venice, Palazzo Papafava, March 13-15, 2002* (ed. T. Rout), Qualital, Italy, 183-190
- [16] Henderson-Sellers, B., Simons, A.J.H. and Younessi, H., 1998, *The OPEN Toolbox of Techniques*, Addison-Wesley, Harlow, UK
- [17] ISO/IEC 15504, 2002, International Standards Organization (in press)
- [18] Saro, D., 2002, A UML profile for platform independent component specifications, internal OOSPICE project team document, 31 January 2002

- [19] OMG, 2001b, OMG, The Software Process Engineering Metamodel (SPEM), revised submission, April 2, 2001, OMG document ad/2001-03-08
- [20] Henderson-Sellers, B., Stallinger, F. and Lefever, B., 2002b, Bridging the gap from process modeling to process assessment: the OOSPICE process specification for component-based software engineering, *Procs. Euromicro Conference*, IEEE Computer Society Press, Los Alamitos, CA, USA

Chapter 9

QCCS: Quality Controlled Component-Based Software Development

Torben Weis¹, Noël Plouzeau², Gabriel Amoros³, Petr Donth⁴, Kurt Geihs¹, Jean-Marc Jézéquel², Anne-Marie Sassen³

¹TU-Berlin, Germany; ²IRISA, Rennes, France; ³SchlumbergerSema, Spain; ⁴KD Software, Czech

Abstract: QCCS is an European IST project that is developing and evaluating a new design methodology for software components. The QCCS methodology simplifies the development process of components that have formally specified non-functional properties. The approach is heavily based on UML because it tries to tackle the problem already during the design phase. QCCS provides a means for modelling contracts aware components and their assembly in UML. Besides the specification process we support the concrete design and implementation of such components.

Key words: Software design, UML, AOP, contracts, quality of service

1. INTRODUCTION

Since software projects became so complex that it took multiple months to finish them, scientists started to search for technologies and methodologies which allowed producing high quality software in time and on budget. Component based software development has the potential to bring us a great step forward in this direction. However, there are several shortcomings in current component models. First of all, components are not well enough specified. Currently their interfaces provide syntactical information about which methods are available and how to invoke them. Any information about the non-functional properties of a component is missing. Classical non-functional properties are time and space complexity of the algorithms used. But in the presence of distributed component systems quality aspects like

security, availability, bandwidth, capacity etc. become more and more important.

In this article we propose to tackle non-functional properties already in the design phase of a component. Other approaches use extensive testing and formal verification. Both are important instruments in achieving high quality components. However, testing can only detect errors. It is of limited help during the design and implementation phase. Formal verification is unfortunately still very difficult – especially for systems of non-trivial complexity – and needs very skilled developers.

Our approach can be subdivided into two steps. First, we need to specify components in a more precise way, which includes clearly tagging the non-functional properties. Therefore, we will use the concept of component contracts. Second, we show how to reuse existing solutions that implement non-functional properties or – if no such solution exists – how to create a reusable one. To achieve this we will employ principles of the AOP: separation of concerns and aspect weaving. We will elaborate the details of AOP in Section 4.

In contrast to formal verification our methodology can not absolutely guarantee that a component actually delivers the quality level that is written in its specification. We need a blueprint that implements the required service level. This solution is parameterised so that it can be applied in similar circumstances. The correctness of the component follows from the correctness of the template solution. The correctness of the template solution can be ensured using testing or verification, code reviews, etc.

The advantage of our approach is three fold. First, it is not likely that developers will make the same mistake twice. Once we have a template solution for availability, security, bandwidth reservation and so on we can easily apply it to new components. These components will then have guaranteed non-functional properties. Second, we separate the design and code that deals with a non-functional property into aspects. That makes the design more concise. Finally, our approach includes specifying the non-functional properties in a formal way. In this way we can determine during deployment whether the functional and non-functional properties of two components allow them to interact correctly.

A special focus is on the ease of use. Our methodology should be easily applicable to different software development processes. Additionally, we want to lower the learning barrier for developers. The methodology should be easily understandable for developers who know about object orientation and modelling using UML.

Let us give a brief overview of this chapter. The next section illustrates how UML can be extended with the concept of component contracts in general. In Section 3 we focus on quality of service contracts and their

representation in UML. Section 4 explains how to implement the specified contracts. Finally, in Section 5 we give a brief summary of our approach.

2. MODELLING CONTRACT AWARE COMPONENTS

Components are often underspecified, which makes their proper reuse a risk in the development process. To remove this shortcoming a more precise specification is needed. Interfaces as we know them from object oriented programming provide a so called functional specification of the component. But there are non-functional issues which have to be specified, too.

Prominent examples for non-functional aspects of a component are performance and security. For instance, a component customer may be interested in knowing the time complexity of a component's computation (e.g. $O(\log(n))$ or $O(n^2)$). Or a component user may want to know whether the component encrypts the data that it sends over the network. Along the same vein, knowledge of bandwidth and latency properties of a component is an important issue for component deployment.

Some conceptual tools have been devised to support these various aspects of component properties: instead of just using component interfaces it is possible to extend them with contracts. A contract (as defined by Bertrand Meyer 0) greatly extends the component specification precision. In 0, contracts are divided into four different levels:

- syntactical contracts,
- behavioural contracts,
- synchronisation contracts,
- quality of service (QoS) contracts.

Interfaces as offered for example by C++ only cover level one. They describe which methods are available and the structure of incoming and outgoing parameters. In Java, the interface may be enriched with synchronisation specifications (level 3). But none of the mainstream object oriented programming languages features solutions for level 2 or 4.

Furthermore, we will make the requirements of a component explicit so that the developer can see as early as possible what it takes to get the component up and running. For example a component implementing an online shop offers an interface for ordering goods. But most likely the component needs some other component that offers a database interface so that the e-shop can store the customer data.

The designer should be able to deduce easily from the component's specification whether the e-shop component has a dependency on the database component. Most currently used component models do not make these dependencies explicit. Component users have to search for this information in

the written documentation (when this information is supplied at all). Good component architectures should not only expose and specify the contracts they offer but also make explicit the contracts they require from others.

2.1 An example

To support our idea we give below a toy design of an e-commerce application. One of its central components is the e-shop component. It offers two interfaces. The `CatalogAPI` interface allows the retrieval of a list of all products, their detailed description and availability and a list of “real” shops. The `PaymentAPI` interface deals with e-payment.

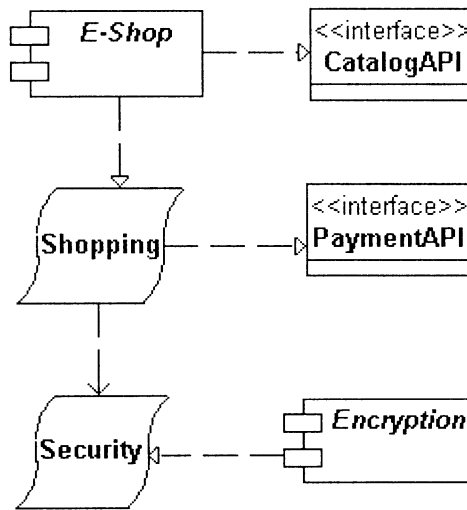


Figure 1. Contract aware components

The `PaymentAPI` interface is valid only in some situations, for instance only if data exchanges can be safely encrypted over the communication link. In our example this is defined by the `Shopping` compound contract that is depicted like a convoluted sheet. It offers the `PaymentAPI` interface (depicted by a dashed line with a closed arrow). On the other side it requires a `Security` contract (depicted by a dashed line with an open arrow). Finally, the `Encryption` component fulfils (offers) the `Security` contract.

One goal of our notation is to stay as close to standard UML as possible. We did not change the notation for components or interfaces. Even the arrow types are standard UML. A dashed line with an open arrow that connects a component with an interface models the fact that the component depends on this interface. A dashed line with a closed arrow between a component and an interface denotes that the component realises this interface. The only new thing developers have to learn is that there exist contracts that can specify

non-functional properties. In fact an interface is just a special kind of contract. However, in order to stay close to the UML notation we did not alter the notation for interfaces.

2.2 Compound contracts

In more complex settings than those in the above example the relationships between *required* contracts and *offered* ones can become quite extensive, especially when level 4 (those with non functional properties) contracts are involved. When dealing with more complicated components we will discover that a set of contracts may have exclusive-or semantics. That means only one contract can be active at a certain time. These contracts often share common subparts. To make this modelling issue more explicit we introduced the concept of compound contracts. In our example the shopping contract is such a compound contract, which is a composition of other contracts. A compound contract can play two different roles. It is either a *required* or an *offered* contract. Another case where compound contracts are useful is the combination of functional and non-functional contracts. For example some component demands a certain throughput for an SQL interface it is using. By grouping the interface and the non-functional contract in a compound contract we can express this relationship.

The exclusive-or semantics can be expressed using UML stereotypes across several dashed lines. In UML this is already done with associations. For example you can model that an instance of some class may participate in one of several associations but not in two of them at the same time. We just applied this concept to contracts.

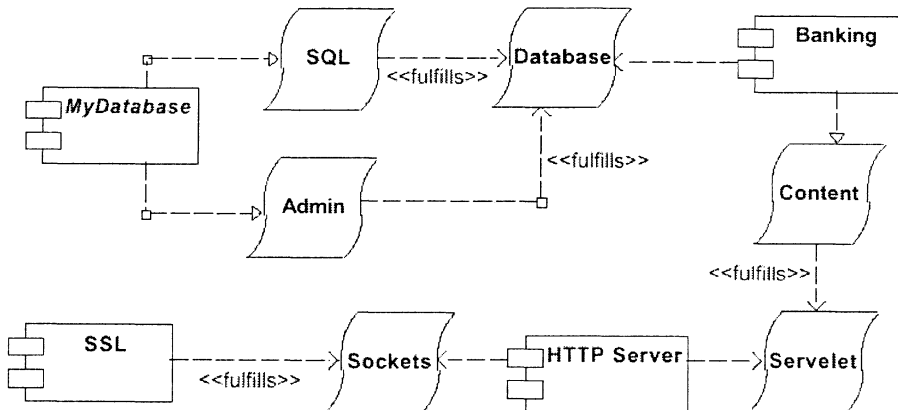


Figure 2. A component assembly

Another reason for using compound contracts is that they enable very high level views of components and their contractual relationships. A real life component will offer a rich set of interfaces and non-functional contracts. Now imagine a diagram with multiple components each offering and requiring a large set of contracts. The diagram will be very large and hard to understand because it is crowded with heaps of boxes and lines.

Grouping contracts that are related in some way or the other is a better approach for abstraction. This grouping can be done using compound contracts. Now it is possible to show in the diagram only the components and the compound contracts. Details that distract the diagram reader are hidden (abstracted) in these diagrams. Nevertheless, the diagram still provides the reader with important information about which components are connected. Figure 2 illustrates how such a diagram might look.

In this figure, the database `MyDatabase` realises two contracts, `SQL` and `Admin`. When taken together they fulfil the `Database` contract of the `Banking` component. This `Banking` component also offers another contract called `Content`. This one matches the `Servlet` contract of the `HTTPServer`. Finally the `HTTPServer` requires some sort of sockets. This contract is directly fulfilled by the `SSL` component. The last observation shows that it is possible to suppress contracts entirely. In this extreme case two components are directly connected by a fulfilment relationship. In terms of the UML the string `fulfils` is just a stereotyped dependency.

2.3 Contract types and contracts

In the previous sections we used the word “contract” with a somewhat vague meaning. However, when we want to model systems with components we need to be more precise. It is crucial to distinguish between contracts and contract types. Contracts and contract types are similar to the concept of objects and classes. Until now we have used the word “contract” where “contract type” would have been more appropriate.

In the static model of Figure 1 we use classes and contract types. *Contracts* are instantiations of contract types and exist at run time. Since the UML provides a means for modelling instances, this allows us to describe the state of a system at a certain point in time or to specify system state evolutions over time. Therefore, we provide a means for modelling contracts in addition to contract types: contract entities coexist with component instances in deployment diagrams.

2.4 Contract negotiation

As we will describe in more detail in the next section it is possible to parameterise contracts. While typical interfaces are not subject to parameterisation during the design and implementation phase, quality of

service contracts (level 4) are just templates that have to be filled at run time with concrete values. For example in a concrete application some component might not be able to ensure availability without knowing its run time environment and this environment's QoS.

This means that contract set up at run time is more than simply connecting an offered contract type instance with another required contract type instance. A concrete contract – the instantiation of a contract type – has to be negotiated between both contracting parties. This can happen at different points in time. Some of these points precede the application launch time: they are part of the design phase, the implementation phase or the deployment phase. A computer aided software engineering (CASE) tool can assist the developer in configuring the contracts.

However, sometimes this static contract computation is not possible, since the quality level that can be offered depends on variables that can change during runtime such as network load, CPU load, memory usage, IO traffic etc. Consequently the concrete contracts can only be negotiated during runtime.

To model this difference we added an attribute associated with every contract that indicates whether the negotiation is static or dynamic. Static means that the contract is configured before the application starts while dynamic contracts are negotiated at runtime.

3. MODELLING QOS-CONTRACTS

In the previous section, we outlined how contracts can be bound to components. However, we did not yet talk about the internals of such contracts. In the QCCS project we focus especially on QoS (level 4) contracts.

3.1 QoS dimensions

Typically a QoS contract type is subdivided into several QoS dimensions. As a first approximation we can think of dimensions as attributes in classes. They have a unique name and an associated type. The reasoning behind the choice of the word *dimension* is that a contract type can be thought of as a vector space. Therefore, the dimensions span a vector space and a contract can be thought of as a point in the vector space spanned by its contract type's dimensions.

When thinking about dimensions in physics we will notice that dimensions typically have a direction. The same applies to QoS dimensions. Their direction indicates in which direction the QoS becomes better. For example bandwidth is better the higher it is. On the other hand latency is better the lower it is. By tagging the direction of every dimension we enable weighting algorithms to choose the best one from a set of values. This becomes

important when a component offers different contracts for one contract type and the other party wants to determine the best contract automatically.

Another observation that can be drawn from physics is that dimensions usually have a unit attached to them. There are two reasons for introducing units. One has to do with scaling. We can use inches, meters, and miles. They have in common that they are units for distances but they differ in a constant coefficient. For our problem it would be enough to allow exactly one of these units so that all parties agree on the interpretation of the numeric value.

The other reason is to attach some sort of semantics to the dimension. When reading something like 500 MB/s we automatically understand that we are talking about some sort of throughput. Units can help developers to understand what a dimension is all about.

Finally, we need to justify that dimensions need a type just like attributes. Types are important even for numerical values since they limit the precision. But there exist some QoS dimensions, which are discrete instead of continuous. For example it does not make sense to demand something like 2.5 replicas of a service. Obviously this dimension should be restricted to integer values. In other situations the type is simply an enumeration.

To sum up: A dimension is a 4-tuple consisting of: name, type, direction, and unit.

3.2 Determining QoS dimensions

An important part of the specification job is to define the contract types. As we will outline in this section it is unlikely that there will ever be one contract type specification for each non-functional – quality of service related – problem.

The whole purpose of contracting is to constrain the minimum quality of service. In order to determine a set of dimensions that allows us to perform contracting we might want to look at the problem of measuring. The delivered quality has to be measurable somehow otherwise it is not reasonable to do any contracting. So it seems to be straightforward to look at the variables we can actually measure. Unfortunately we can not deduce the dimensions of a contract type entirely from these variables. The values of the variables describe the quantified QoS that is actually delivered by the service. In the ideal case a diagram showing a measuring-variable together with the time axis will show a horizontal line. In this case the delivered quality is always exactly the quality negotiated between client and server. Unfortunately we do not have these horizontal lines in real life environments.

So we have to define in a contract what we still consider to be acceptable and what we have to reject. From a mathematical point of view there are an unlimited number of possibilities for specifying these constraints. Some mathematical concepts are frequently used in this situation. We could describe

minimum boundaries, which must not be crossed. Or we can demand that the average quality measured over a certain time range must not fall below a certain value.

However, in other situations different constraints may be useful. For instance, one may be interested in the QoS availability category. We can determine whether the service is actually available or not. Unfortunately our measure can be based on a boolean variable only: either the service is available for our requests or it is not. Nevertheless we can have different demands:

- the service must not be continuously unavailable for longer than t minutes;
- the accumulated downtime in one month must not be longer than 10 hours but no more than 5 hours per day;
- the service has to be continuously available for at least t hours per day.

There may be many possibilities, some of them are really useful ones in certain situations. In practice, this high number of possibilities may limit the interoperability. If everybody defines his own set of contract types then we may have a hard time to smoothly integrate *commercial off the shelf* (COTS) components in the development and deployment process. We are confident that it is possible to find definitions that satisfy the majority of cases.

Another problem that we encountered is that it is sometimes hard to sample and compute one of the measuring variables. An example is security. It is actually not possible to measure security in the same way as performance, bandwidth and the like. So we use values which are more closely bound to the mechanism that is used to provide a certain QoS category. Possible dimensions are the key length used for encryption or the encryption algorithm used.

3.3 QoS contract types

A contract type is a composition of several dimensions. However, grouping a set of dimensions is not the sole purpose of a contract type. Its main contribution is to assign semantics to the dimensions. There are many possible ways of expressing semantics. For example we could try to model the semantics using state charts and interaction diagrams. Or we could use a formal specification language. In our approach it does not matter how the semantics are specified as long as they are precise. The reason is simply that none of our tools will read the semantics. It is up to the developer to read them, understand them and provide a correct implementation. Other approaches that are based on testing or formal verification do of course require a notation of semantics that can be understood by a tool. We believe that the essential requirement is that the component developer and the person deploying the component agree on the semantics.

Figure 3 shows our notation for contract types [2]. The notation closely follows the UML notation for classes [6]. The major difference is that the shape of a component is convoluted. The dimensions are listed in the compartment below the contract type's name.

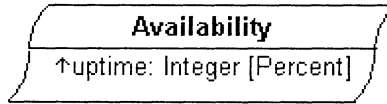


Figure 3. Notation for contract types

The notation for dimensions resembles partially the UML notation of attributes with some extensions. The arrow indicates the direction. In the above example higher values are considered to be better than lower ones. Then follows the name and the type that is used to store values of this dimension. On the right hand side in brackets you can find the unit of the dimension. The possible units are defined by an enumeration.

3.4 Instances of contract types

The notation for contract type instances (in short: contracts) resembles the UML notation for objects. The first line shows the name of the contract and – separated by colon – the name of the corresponding contract type. This string is underlined like the names of objects in UML.

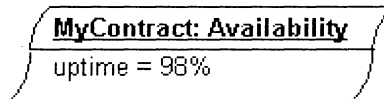


Figure 4. Notation for contract type instances

The compartment below the name holds all dimension values. The first string is the name of the corresponding dimension followed by a textual representation of the value. Finally, the unit of the dimension is shown.

4. REALISING CONTRACTS

Until now we discussed how to model contract aware components using UML. This results in a more precise specification of components and component assemblies. However, at some point in time these components have to be implemented and we want to be sure that they actually deliver the service level that their specification promises.

As outlined in the introduction some well established approaches in this area are testing and verification, but they have several shortcomings. They can not help during the design phase and they are quite complex to apply to real scale designs. Our approach is to reuse existing solutions, which have guaranteed properties. In an ideal case an automated tool can read in the specification and produce a skeleton that already implements the non-functional properties specified in the component's contracts.

4.1 Cross-cutting, scattering and AOP

The major problem that needs to be solved here is the cross-cutting and scattering. The design pieces and code fragments that realise a certain non-functional property are usually scattered across the entire design and code. It is usually not possible to gather these pieces in one class. Just think about the availability example presented above. In order to realise availability we need a lot of design and code. First of all we need a mechanism that can realise availability. Let us consider some kind of fault tolerant load balancing. We need to use the concept of replicas and a sequencer. This will for example require some work in the area of the middleware. However, that is not enough. In addition we will need some algorithms that perform the negotiation at runtime because the quality of service needs to be monitored. Finally, a customer may have to pay for the service delivered by the component. In this case a link to the billing facility of the company is needed.

This example shows that we will have to handle concepts in completely different areas of the design. This is exactly the problem of *cross-cutting and scattering*. Therefore, it is almost impossible to separate these pieces using standard object oriented techniques.

A modern solution to this problem is *aspect orientation*. The major benefit of aspect orientation is the separation of formerly scattered pieces into aspects. Aspect oriented programming (AOP) allows us to group all design pieces and code fragments that deal with the realisation of a certain non-functional property in one place. This is the major prerequisite for reuse, because a designer cannot reuse something in a new design that can not be clearly isolated in an existing design.

However, there is a price to pay for this advantage. Humans find it very convenient to work with design and code that is structured by the principles of aspect orientation, since the entire structure becomes clearer. On the contrary, most software tools do not take aspect separation into account. Therefore, it is necessary to invert the process of separation. This process is called aspect weaving. Once the aspects are woven into the remaining design and code we can continue using our existing tools such as compilers and interpreters.

The major challenge in every aspect oriented approach is to build a good aspect weaver. Since we still target the design phase of a component we will have to provide an aspect weaver that can weave UML models.

4.2 Reusable aspects

Aspect orientation brings us a great step forward in our attempt to isolate code and design pieces. However, AOP driven technologies like AspectJ 0 do not foster reuse of aspects in a different context, because that is not the original goal of AOP. Instead it is possible to exchange one implementation of an aspect with another one. That is the opposite of what we want to do. We intend to use one aspect for several targets while classical AOP uses different aspects for one target.

Consequently it is not surprising that a simple port of the AspectJ ideas to UML won't be enough. The main problem is that aspects contain information that tells the aspect weaver where they bind to their target. This is out of question for our approach since that contradicts our idea of aspect reuse for different targets. One conclusion that can be drawn from this is that the aspects may not contain any information directly related to their target.

Obviously, some information about the correspondence of target and aspect is required, otherwise the aspect weaver can not know how to weave the aspects into the target. The basic idea is to model so called *aspect-invocations* that describe where to weave which aspects into the target. In this way we can use an aspect very much like a function. We can invoke it and pass as arguments the points in the target where the aspect is to be woven in. To sum it up, we end up with three categories: the target, the aspects and the invocations which bind both together.

4.3 Aspect signatures

Since we consider aspects as functions we have to deal with the signature of an aspect. A function invocation passes a set of arguments to the invoked function. These arguments must comply in type and structure to the signature of the function invoked. Consequently, we have to provide a means for defining signatures of aspects. This is not part of AspectJ since these aspects know by themselves where they have to be woven into the target. Logically, there is no need to pass arguments and consequently no need for a signature.

Function signatures can become almost arbitrary complex, as languages like C++ show. We have to examine how much complexity is actually needed. The least two things that a signature should describe is

1. how to structure the arguments;
2. the kind of argument expected.

The most simple argument structure is a simple list. The other extreme is an arbitrarily shaped graph. We investigated a set of examples to determine our requirements. We concluded that the simple list structure is not at all sufficient. Suppose we have to pass to an aspect a set of classes and a subset of each class' methods as arguments. By using plain parameter lists only it is impossible to pass this information to the aspect. We decided to allow arbitrarily shaped trees as argument structures. This is very flexible but still much easier to deal with when compared with graphs.

The second topic of interest is the type of a parameter. In object oriented languages this type is a combination of a class and sometimes additional information that tags a parameter as pointer, array or reference. The types of arguments we want to pass to our aspects are elements of a UML model such as methods, classes, states of a state chart and so on. Consequently, we have to refer to the UML meta-model. The meta-model describes which concepts are provided for modelling. Such concepts are classes, methods, parameters, states, state transitions and the like. Obviously this corresponds exactly to our parameter types. The type of an aspect parameter is given by a UML meta-class. Additional concepts like arrays are not needed since they can be expressed in the tree structure. The following examples show what the parameter list (or signature) of an aspect can look like.

```
MyClass: Class, OtherClass: Class
Iface: Class, {Op: Operation}
Iface: Class, {Get: Operation, Set: Operation }
MyClass: Class, {Op: Operation, {P: Parameter}}
```

The first signature above describes that the aspect expects as arguments two classes of the UML target model. The second one features parameters with a 1:n relationship which is expressed by the curly braces. For every class we may pass a set of operations. The third signature declares that we may pass a set of tuples for each class. Each tuple must contain exactly two operations. Finally, the fourth signature shows that parameters may be nested arbitrarily deep. In this way we can pass arbitrary tree structures as arguments to an aspect.

Now we want to look at the invocation of an aspect and how to specify the arguments that are passed along with every invocation. Assume a target model as specified in Figure 5. Furthermore, we assume the following aspect signature:

```
C: Class, {O: Operation}
```

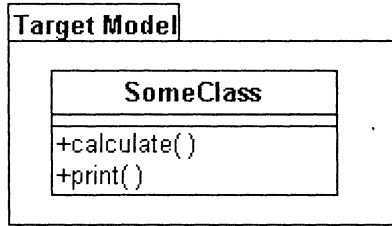


Figure 5. Target model

A possible invocation of the aspect can look like this:

```
(SomeClass, {calculate, print})
```

This will assign `SomeClass` to parameter `C`. The two methods `calculate` and `print` will be mapped to the parameter `O`. Another way of thinking about this is to consider the arguments as a tree. `SomeClass` is the root node and the two methods are the leaves of the tree.

4.4 Aspect notation

Our notation for aspects resembles the standard UML notation for packages. The signature appears in the upper right corner in a box surrounded by a dashed border. UML users will be familiar with these sort of boxes. They are used for template classes in UML, too. These visual similarities are used on purpose, since the signature is some sort of template parameter. However, there are major differences between our aspects and UML class templates. The two most obvious differences are that UML templates may only have simple parameter lists and do not at all support the concept of wildcards.

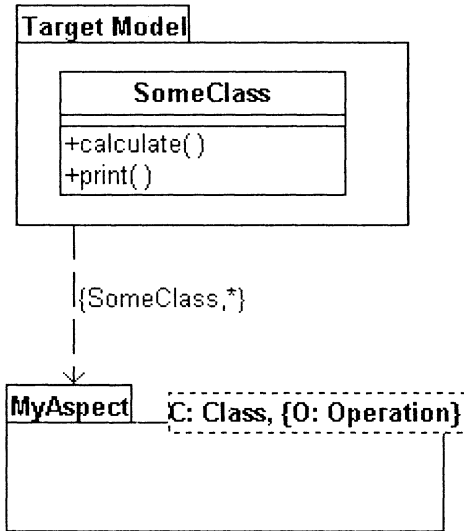


Figure 6. Aspect notation

Figure 6 shows a small example. We can see one invocation of the aspect. The invocation is modelled as a stereotyped dependency between the target model and the aspect. The arguments are written on top of the invocation arrow.

The example does not show what is inside the aspect. We are working on an UML based description for aspects and a corresponding aspect weaver. Modelling the aspect interior is then almost trivial if you already know how to use the UML.

5. SUMMARY & CONCLUSIONS

We presented a way of specifying contract aware components. In doing so we placed a special emphasis on quality of service contracts. For this purpose we extended the UML with several stereotypes and some new notation. Our approach does not unconditionally require the new notation, but it significantly helps to increase the readability of the diagrams. Standard UML tools can nevertheless be used to model the same things. The only drawback is that the developer will face the UML stereotypes directly. In our modelling tool these are hidden behind the new notation.

Once the specification phase is finished we have to deal with standard class diagrams, state charts etc. A generator produces a model skeleton from the specification. If the generator knows of some aspect that can realise a certain contract type then it would integrate the correct invocations of this

aspect into the skeleton. Otherwise it generates an empty aspect and it is up to the developer to provide a first solution for this contract type. The generator is not very complex and it needs only minor (if any) modifications to integrate new aspects for new contract types. In this way it is very easy for development teams to extend their toolset with new aspects.

In this chapter we explained how to declare aspects and how to model their invocation. However, we did not talk about how to define an aspect. There are different possibilities for doing so. We decided to stay with the UML. This means the interior of an aspect looks very much like a normal UML diagram with a few extensions.

We think that the QCCS approach brings us a great step forward in the direction of quality controlled software engineering. It is a very practicable approach, which has its merits and shortcomings. The major limitation is that the first implementation of a contract type needs to be correct. However, the QCCS methodology does not provide a means for proving the correctness of this implementation. On the other hand one contract type is usually used in many different projects. Thus, the chances of reuse are enormous. This is especially important, because the QCCS methodology can guarantee the correctness of all components with respect to a certain contract type as long as the single aspect is correct. Instead of verifying each single component it is enough to verify the single aspect.

REFERENCES

- [1] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.. *An overview of AspectJ*. In Proceedings of the European Conference on Object-Oriented Programming, Budapest, Hungary, 18-22 June 2001.
- [2] Weis, T, Becker Ch., Geihs, K. and Plouzeau, N., *An UML Metamodel for Contract Aware Components*, Proc. of UML'2001 conference.
- [3] Meyer, B., *Applying Design by Contract*. IEEE Computer Special Issue on Inheritance and Classification, 1992. **25**(10): p. 40-52.
- [4] Beugnard, A., et al., *Making Components Contract Aware*. IEEE Computer Special Issue on Components, 1999. **13**(7).
- [5] Frolund, S. and J. Koistinen, *Quality of service Specification in Distributed Object Systems*. Distributed Systems Engineering Journal, 1998. **5**(4).
- [6] Object Management Group, *OMG Unified Modeling Language Specification*, Version 1.4, September 2001.

Chapter 10

Components for Embedded Devices

The PECOS Approach

Thomas Genssler, Alexander Christoph, Michael Winter and Benedikt Schulz
FZI, Forschungszentrum Informatik, Germany

Abstract: Software is more and more becoming the major cost factor for embedded devices. Already today, software accounts for more than 50 percent of the development costs of such a device. However, software development practices in this area lag far behind those typically applied in the information systems development domain. Reuse is hardly ever heard of in some areas, development from scratch is common practice and component-based software is usually a foreign word. PECOS is a collaborative project between industrial and research partners that seeks to enable component-based technology for a certain class of embedded systems known as "field devices" by taking into account the specific properties of this application area. In this paper we introduce a component model for field device software. Furthermore we report on the PECOS component composition language CoCo and the mapping from CoCo to Java and C++. We conclude by giving an overview on the PECOS software development process.

Key words: Embedded devices, field devices, software from components, composition language

1. INTRODUCTION

The state-of-the-art in software engineering for embedded systems is far behind other application areas. Software for embedded systems is typically monolithic and platform dependent. Development from scratch is common practice. The resulting software is hard to maintain, upgrade and customize. Beyond that it is almost impossible to port this software to other hardware platforms. Component-based software engineering is expected to bring number of advantages to the embedded systems world such as faster

development cycles, the ability to secure investments through reuse of existing components, and the ability for domain experts to interactively compose and adapt sophisticated embedded systems software out of pre-fabricated parts. The key technical questions and challenges are:

- **Component model:** What kind of component model is needed to support modularization and re-use of software for embedded systems? Which non-functional aspects of this software (such as timing constraints) have to be modeled explicitly to enable automated compositional reasoning?
- **Lightweight composition techniques:** How can component-based applications be mapped on efficient and compact code that fulfils the hard requirements imposed by the application domain?
- **Platforms and tools:** How can we increase software portability (and thus increase re-use and productivity)? What tools are needed to support efficient specification, composition, validation, and deployment of embedded systems applications built from components?

The PECOS project aimed to enable component-based software development for embedded systems. To achieve concrete results, PECOS was focusing on a specific area of embedded devices, so-called *field devices*. The results of the project included a process model for component-based software development, a component model that addresses the specific needs of the application area as well as the necessary method and tool support. A real case study device has been developed to evaluate the concepts and to put the approach into practice.

Section 2 introduces the PECOS case study, summarizes the particular requirements of field devices for a component-based software development (CBSD) approach, and provides an example application that illustrates the PECOS working domain. Section 3 introduces the PECOS (field device) component model. In Section 4 we introduce the PECOS composition language for component based software. Examples are used to illustrate the concepts and the mapping between the language and the model. Section 5 provides concepts for deploying the specified software to *real* devices. It describes how components are mapped to target code and how certain concepts are implemented. Section 6 presents the PECOS software development process.

2. CASE STUDY DESCRIPTION

ABB's instruments business unit develops a large number of different field devices, such as temperature-, pressure-, and flow-sensors, actuators and positioners. A field device is an embedded system with hard real-time constraints. Field devices use sensors to continuously gather data, such as temperature, pressure or rate of flow. They process this data, and react by

controlling actuators like valves or motors. To minimize costs, field devices are implemented using the cheapest available hardware that is sufficient to achieve the task. A typical field device may contain a 16-bit microprocessor with only 256KB of ROM and 40KB of RAM.



Figure 1. Pneumatic Positioner TZID

The software for a typical field device, such as the TZID pneumatic positioner shown in Figure 1, is monolithic, and is separately developed for each kind of field device. This results in a number of problems.

- **Little code reuse:** The same functionality (e.g., non-volatile memory-manager, field-bus driver, or control-algorithms) is re-implemented at different development sites in different ways for different field devices.
- **Plug-incompatibility:** Functions and modules are implemented for a specific device without standardized interfaces.
- **Inflexibility:** Monolithic software is hard to maintain, extend or customize.
- **Cyclic execution model:** The software system often comprises several cyclically running tasks with different cycle times (e.g., 5ms, 10ms and 50ms). This makes it hard to incorporate sporadic and long running functions without introducing deadline failures, that is a deadline is not met when it should be. Furthermore it is error-prone to change the execution model from cyclic execution to process-based scheduling [2].

In order to demonstrate the applicability of the CBSD approach for embedded systems, the PECOS project is developing the hardware and software for a demonstrator field device. The task of this PECOS demonstrator field device is to control a three-phase motor that is used to close or open a valve (see Figure 2).

The motor is driven by a frequency converter that can be controlled by the field device over a Modbus connection (an industrial communication protocol). The motor is connected to the valve either directly via a worm shaft or using additional gearing (4). A pulse sensor on the shaft (5) detects the speed and the direction of rotation. The PECOS demonstrator field device (1) is equipped with a web-based control panel (7) with some basic elements for local operation and display. The demonstrator device can be integrated in a

control system via the field-bus communication protocol Profibus PA (6). The device is compliant with the Profibus specification for Actuators [3], [4].

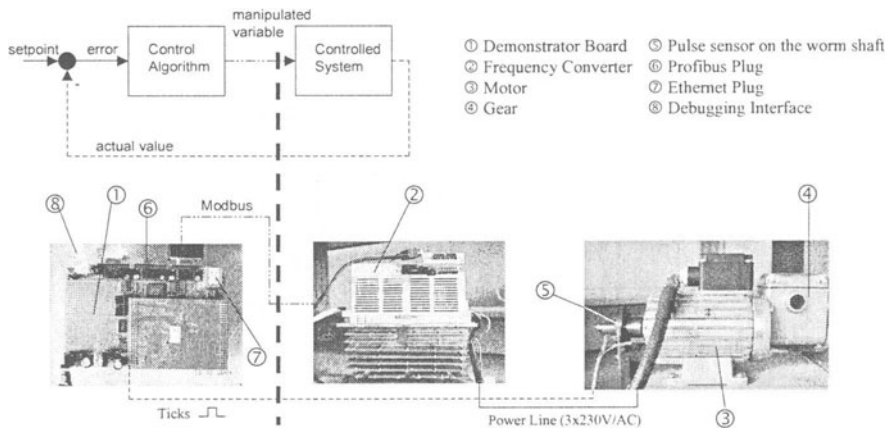


Figure 2. PECOS Case Study Device

We will use the following example throughout this chapter to illustrate the PECOS component model and composition language. A part of the PECOS case study is concerned with setting a valve at a specific position between open and closed. Figure 3 illustrates three connected PECOS components that collaborate to set the valve position; the desired position is determined by other components not shown here. In order to set and keep the valve at a certain position, a control loop is used to continuously monitor and adjust the valve. This `PositionValve` control system consists of three components:

- The component `ProcessApplication` obtains the desired position of the valve (Set-Point) and reads the current state of the valve from the `FQD` component. This information is then used to compute a frequency for the motor. Once the motor has opened the valve sufficiently, ascertained by the next reading from the `FQD`, the motor must be slowed or stopped. This repeated adjustment and monitoring constitutes the control loop. The component `ProcessApplication` runs *cyclically* and *synchronously* with a defined cycle time and a worst-case execution time which must not be exceeded.
- The `FQD` (Fast Quadrature Decoder [5]) component is responsible for capturing events from the motor. This component abstracts from a micro-controller module that does `FQD` in hardware. It provides the `ProcessApplication` with both the velocity and the position of the valve. The `FQD` component runs *asynchronously* whenever the respective motor event occurs.

- The ModBus component acts as an interface to the frequency converter that determines the speed of the motor. The frequency to which the motor should be set is obtained from the ProcessApplication component. ModBus outputs this value over a serial line to the frequency converter using the ModBus protocol [22]. Since communication over ModBus is rather slow and does thus not fit into the cyclic execution scheme.

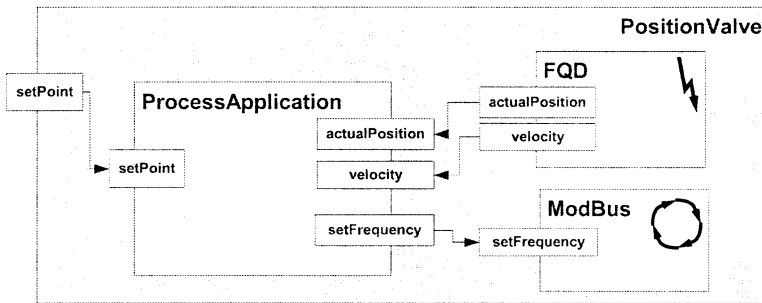


Figure 3. FQD Control Loop

This example illustrates the key issues – besides the tight resource situation – concerning the field device domain:

1. **Cyclic synchronous behavior:** Most components responsible for a single piece of functionality are repeatedly executed synchronously (by a scheduler) with a specified cycle time and a worst-case execution time. ProcessApplication is an example of this. The execution must not take longer than the specified worst-case execution time.
2. **Threading and asynchronous execution:** Some components do not fit into the cyclic synchronous scheme are passive, while others (like FQD or ModBus) have their own thread of control in order to react on asynchronous events or to perform long-running computations in the background.
3. **Data-flow-oriented interaction:** Components communicate by means of data that flows between the entities of the system. The interface of a component consists of a set of data ports. The dataflow is usually implemented using shared memory.

3. A COMPONENT MODEL FOR EMBEDDED SOFTWARE

In order to apply CBSD techniques to embedded systems software, we must be precise about what we mean by a *component*. In particular, we must take care to specify how components are *structured* and *composed*, which

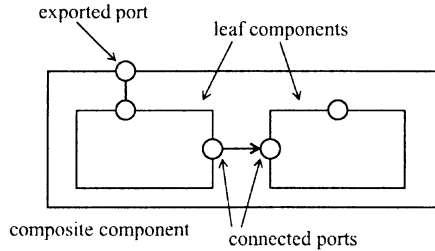


Figure 5. A Composite Component

The model is able to represent three types of components that are relevant in the embedded systems domain.

Passive Components (e.g., *ProcessApplication*) encapsulate some functionality: when they are executed (scheduled), they may read values from their input ports, perform some calculation (depending on these values as well as on their internal state) and write values to their output ports. Typically, they execute on a regular basis (e.g. every 10 ms). Their execution characteristics can thus be characterized by a *cycle time* and a *worst case execution time*.

Active Components (e.g., *ModBus* in Figure 5) are called *active* because they run in their own thread of control. Active components are used for two purposes. First, they model long-running, continuous activities. Second, *composite* active components are responsible for *scheduling* their constituent components in a way that respects the deadlines imposed by the real-time constraints. In addition, a complete system composed of components is also modeled as an active composite component.

Event Components (e.g., *FQD*) are components whose functionality is triggered by an event. They are typically used to model hardware elements that generate events. Typical examples are timers, used to keep track of deadlines, or devices that emit events that encode status information, such as the current rotation speed of a motor, the current temperature, and so on.

Components are characterized by their *properties*, which encode information such as timing (e.g. their cycle time) and memory usage or other non-functional information about a component.

3.2 Execution Model

In addition to the static structure described above, the PECOS model comprises an execution model that describes the behavior of a field device software. By using Petri nets [7] to represent the execution model, we set a formal basis to reason about real-time constraints and automatically generate real-time schedules for software components.

The execution model deals with the following two issues: **Synchronization** – how to synchronize data-flow between components (esp. between components that live in different threads of control) and **Timing** – how to make sure that the functionality of a component is executed according to its cycle time and specified deadlines.

A composition of components has as many threads of control as there are active components. Each composite active component is responsible for scheduling the passive components under its control and to take care of synchronization with the active components it contains. This is carried out by the composite active component by calling the `execute()` and `sync()` methods (see Section 4.2) of its constituent passive and active components, respectively. The sequence of calls to these functions is pre-determined by a static schedule, which is computed from the real time constraints specified for the system. The feasibility of the schedule is checked using techniques like RMA ([23]).

We formalize the execution semantics of the PECOS model by means of a Petri net interpretation. Three types of tokens are used in this Petri net model to represent data-flow, control flow and synchronization, respectively. The details of this formalization, however, are beyond the scope of this chapter. They are described in [8], [9].

4. THE COCO COMPONENT LANGUAGE

In this section, we introduce the component language *CoCo*. The CoCo language is the syntactical representation of the model described in Section 3. The language is intended to be used for 1) the specification of components, 2) the specification of entire field device applications and 3) the specification of architectures and system families. In addition, CoCo supports reuse of components and architectures and facilitates compositional reasoning. Moreover, CoCo serves as input for scheduler computation and code generation.

We first give an overview on the CoCo language. We describe how developers can specify components in terms of their interfaces and their behavior. We subsequently show how CoCo supports the specification of system families. Finally, we sketch how CoCo specifications can be used for reasoning about functional and non-functional properties of a system. A more detailed description of the CoCo language is given in [11].

4.1 CoCo Language Overview

Components represent units of computation and are the major means of structuring a CoCo system. CoCo supports all component types of the

component model. For example, Figures 6 and 7 show our running example in CoCo. In Figure 7, we see an active component (marked with the keyword `active`), an event component (keyword `event`) and a passive component. In analogy to the OO model, components play the role of classes. Components define a scope in the same sense as OO classes do. Components can be instantiated that is, there can be several instances of a component at run-time, each of which with an unique identity. The component `PositionValve` in Figure 7, contains instances of other components (e.g., of component `FQD`). However, instances have to be declared statically that is, there is no mechanism for dynamic creation of new instances at run-time (i.e., using some sort of *new* statement known in standard OO languages).

Programming in CoCo is data-flow-oriented. **Ports** (e.g., `setPoint`) denote data flow into or out of a component and they are the only means to communicate with a particular component. One can think of the set of ports of a component as the interface to a piece of functionality or behavior that is executed cyclically or in response to a certain event in order to compute output values depending on the current input values and/or the internal state of the component. The actual implementation of the behaviour, however, is not specified at the level of CoCo specifications but added to the component as C++ or Java code (see Section 4.2). The only information available about this implementation is the worst-case time it takes to perform the computation (property `execTime`) and the interval between this computation (property `cycleTime`). These values are specified in CoCo as component properties. Ports are assigned both a data flow direction (`input`, `output`, or `inout`) and a data type. Furthermore, ports can be declared mandatory (default) or optional. The former means that this port must be connected for a component to be used correctly. Failure to connect all mandatory ports of a component will result in an error. Optional ports need not necessarily be connected. Often they only provide access to additional status information about a component.

```

event component FQD {
    // out ports
    output float actualPosition;
    output float velocity;
    // properties
    property cycleTime = 100;
    property execTime = 10;
}
active component ModBus {
    // in ports
    input float setFrequency;
    // properties
    property cycleTime = 100;
    property execTime = 10;
}

component ProcessApplication {
    // in ports
    input float setPoint;
    input float actualPosition;
    input float velocity;
    // out ports
    output float setFrequency;
    // properties
    property cycleTime = 100;
    property execTime = 20;
}

```

Figure 6. The FQD Control Loop Components Specified in CoCo

Components are interconnected through the use of **connectors** (e.g., connector `c1` in component `PositionValve`). Connectors connect a list of ports

defined either in the current component (like port `setPoint` in connector `c1`) or by one of the contained instances (that is, instances in the same scope). Different connectors that share a common port represent the same connection. For composite event and active components, this is only true within the scope of their parent component while for passive components this also holds for ports of instances within this particular passive component.

```

component PositionValve {
  ModBus mb;
  FQD fqd;
  ProcessApplication pa;
  input float setPoint;
  property execOrder = "pa, fqd, mb"; // execution order
  connector c1 (setPoint, pa.setPoint);
  connector c2 (fqd.actualPosition, pa.actualPosition);
  connector c3 (fqd.velocity, pa.velocity);
  connector c4 (pa.setFrequency, mb.setFrequency);
}

```

Figure 7. The FQD Control Loop Specified in CoCo

Properties serve to specify functional and non-functional features of a component, such as initialization values for ports, memory consumption and worst-case execution time. They can be structured in so-called *property bundles*. These bundles group properties that semantically belong together, such as scheduling information (worst-case execution time, cycle time). Properties can be used by tools to inspect the component in different phases of the development process (e.g., by using execution orders and times when calculating a schedule). Properties can be set on a per-component basis and a per-instance basis.

4.2 Adding Behavior to Components

CoCo does not only support the specification of component interfaces but also provides some help for the specification of the actual implementation of the behavior of components. There are two ways of adding behavior to a component. One is by composing a component out of existing components. However, the implementation of the basic behavior of leaf components is not specified in CoCo directly but a general purpose programming language is used. PECOS supports Java and C++ (including Embedded C++, [16]) for this purpose. To fill in implementation code written in one of these languages, CoCo provides three pre-defined hooks:

- **Initialize:** Initialization code for a particular component such as init values for ports is added here. This code is executed by the run-time system upon start-up.
- **Execute:** Serves to specify the actual functionality of a component that is, the algorithm that computes output values using the internal state of a component and/or input values. The time of execution of this functionality

depends on the component type. For a passive component this functionality is invoked synchronously by a scheduler. For active components, this functionality is executed continuously within a separate thread. Event components (including timer components) perform their functionality when the event occurs that this particular event component listens to.

- **Sync:** Active and event components have this additional part. The code in sync is executed synchronously – like execute of passive components – by the scheduler and serves to exchange data between the asynchronous thread or event handler of active components respectively event components and the synchronous outside world.

The code that can be filled in at these three hooks has to be valid target language code (C++ or Java). A developer can only use primitives defined by the PECOS run-time environment [21]. This means in particular that a developer cannot start new threads or do anything else which might affect schedules. To deploy a component to a particular target platform (i.e., to generate code for this platform) all hooks of each participating component must be filled in with appropriate target language code. The code generator adds these code fragments to the generated code. Code generation is discussed in more detail in Section 5.

4.3 Specifying Software Families with CoCo

Components define the concrete parts that make up the overall system. To specify architectural styles or families of components or families of entire applications (devices), however, some additional support is needed. CoCo provides the concept of abstract components for this purpose. Abstract components represent templates of entire systems, or frameworks, that may be filled in later on with concrete components. Abstract components do not have a representation in the model as they do not contribute to the run-time behavior of field device software. They are merely a technique to simplify specification and to enable reuse of designs.

```
abstract component PecosControlLoop {
  role AbstractProcessApplication PecosPA;
  role AbstractControlDevice PecosCtrl;
  role AbstractFeedbackDevice PecosFdbck;
  input float setPoint;

  connector setPoint(setPoint, PecosPA.setPoint);
  connector feedback1(PecosPA.actualPosition,
                    PecosFdbck.actualPosition);
  connector feedback2(PecosPA.velocity, PecosFdbck.velocity);
  connector control(PecosPA.setFrequency, PecosCtrl.setFrequency);
}
```

Figure 8. Definition of an Architectural Style

Besides the elements known from normal components, abstract components can define so-called *roles*. Roles are typed variation points or holes in a (micro-)architecture. Figure 8 shows the specification of an architectural style for control loops.

We assume that a `PecosControlLoop` should always have an instance of sub-type of `AbstractProcessApplication` that plays the role `PecosPA` in our valve controller architecture. `AbstractProcessApplication` again is an abstract component that defines a certain interface (i.e., ports, properties) every process application component has to conform to. Thus, roles serve as placeholders for instances. These placeholders can also be connected by connectors as if they were normal instances. This way a developer is able to specify an entire family of applications that share a common architecture in terms of the components involved and their data-flow dependencies. To create a specific member of this family, a component has to implement the respective abstract component. Implementing an abstract component means that all roles defined by this abstract component have to be bound to suitable instances and that all connectors, instances, ports, and properties defined in this abstract component become now part of the implementing component. In our example the role `PecosPA` is bound to an instance of component `ProcessApplication`. The component `ProcessApplication` on the other hand is required to implement the abstract component `AbstractProcessApplication`.

4.4 Composition Checking

In order to be valid, a composition must follow certain rules. Besides simple syntactic rules, that are checked by a composition language parser, some semantic rules must also be followed. These rules express requirements, that emerge from the component model such as “if a component implements an abstract component, it must bind all roles” or “all mandatory ports must be bound”. First order predicate logic is used to check these rules. The PECOS composition tool is able to generate a set of Prolog facts out of a composition. These facts describe the whole system, together with all included components and their connections. Semantic rules are formulated as Horn clauses, which are checked against the generated facts [17].

Besides semantic rules imposed by the PECOS component model, application domain specific rules may be imposed on a specification. For example, embedded systems in a particular domain, e.g. field devices, have a specific set of requirements that could be checked using composition rules.

Finally, it may be important to impose application specific or project specific rules. Such rules could express certain requirements for debugging or release versions of the software, dependencies between components, platform specific property settings, etc. When checking these rules, they are formulated as Prolog queries which in turn are validated against the generated facts.

5. CODE GENERATION: FROM COCO TO C++ AND JAVA

CoCo can be used to specify a system using the PECOS model. Additionally, we have a language mapping from CoCo to target languages such as C++ and Java in order to be able to build a functioning system out of a CoCo specification. This section briefly introduces the basics of this mapping.

5.1 Mapping Components

Components in the PECOS model are directly mapped to classes in the target language. Components are functional units that perform some actions by means of an execute part. Passive components within the same scope perform their execution synchronously, one at a time. Active components, however, run in parallel with the rest of the system. This is realized by mapping the PECOS model to a prioritized, pre-emptive multi-threaded system (the PECOS Execution Environment). For assigning components to a particular thread we apply the following two rules:

1. Every active and event component runs in its own thread.
2. Every passive component that is part of a composite runs in the same thread as its direct parent.

Instances of components are mapped to objects in the target language. More specifically, since instances of components can only occur inside of composite components, these instances are mapped to member variables. Instances are given the same name as the name in the specification and can be accessed through the class representing the composite component.

```
package org.pecos.generated;
import pecos.rte.component.PecosPassiveComponent;
import org.pecos.generated.DataStore;

public class PositionValve extends PecosPassiveComponent {
    public PositionValve() {
        super( "PositionValve" );
    }
    public ModBus mb = new ModBus();
    public FQD fqd = new FQD();
    public ProcessApplication pa = new ProcessApplication();
    public float get_setPoint() {
        return (DataStore.val_float[DataStore.PositionValve$setPoint]);
    }
    public void initialize() {
        mb.initialize();
        fqd.initialize();
        pa.initialize();
    }
    public void execute() {
    }
}; /* PositionValve */
```

Figure 9. Generated code for the PositionValve Component

The PECOS model defines ports as the only feasible way for exchange of data between components. The model determines three different types of ports: `input-ports`, `output-ports` and `inout-ports`. In the target language, ports are represented as `getter` and `setter` methods depending on the type of port. `Input-ports` are mapped to a `get-method`, `output-ports` to `set-methods` and `inout-ports` are mapped to both `get-` and `set-methods`. When we map ports to `get-` and `set-` methods, we are able to hide the implementation of connectors between the ports in the methods and, at the same time, give the user an easy means of accessing the ports. As an example, Figure 9 shows the generated code for the `PositionValve` component.

5.2 Mapping Connectors

CoCo is designed to specify a system that is consisting of pair-wise interacting components, instead of merely being able to define stand-alone entities. Therefore a data exchange mechanism has been introduced that connects ports through connectors. A connector ensures that data at an output port is "moved" to the connected input port.

Data exchange (DS) between components can be achieved in various ways. Two common approaches are: shared memory and copying values. In the language mapping that we describe here we have chosen a hybrid approach that incorporates both strategies. Within a collection of components running in a single thread data is exchanged through shared memory. This is achieved by assigning a shared memory page to every thread for data communication.

The data store that implements the shared memory is an automatically generated artifact. Inside a single thread, mapping connectors under the PECOS model to the Data Store is straightforward. The value of a connector is stored at a predetermined location in the Data Store. The generated `get-` and `set-methods` to exchange data on a port use indices that read from, respectively write to, that location in the Data Store. To achieve this, the generated classes use constant indices that are used to access the right value. When two ports are connected they use exactly the same index (the constant variable has the same value). This results in two ports that use the same location in the DS.

Active components together with their passive subcomponents are running in their own threads. Each thread has its own Data Store for the connectors local to this particular thread. Connectors between ports that belong to components in different threads work differently. They exchange data by copying values from the Data Store in one thread to the Data Store in the other thread. A scheduled synchronization method `sync` (see Section 4.2) is used to exchange data between the different threads.

Since the behavior of the `synchronize` method cannot be known beforehand and it is not specified in the model either, users currently have to provide their own methods. A typical example of a `synchronize` method would copy data in or out of the thread's data store depending on the state of the component. Therefore, two utility methods are generated to aid in performing these tasks:

- `import_<portname>`: Imports the value from the “outside” world into the “local” DS (input- and inout-ports only).
- `export_<portname>`: Exports the value from the “local” DS to the “outside” world (output- and inout-ports only).

5.3 Executing The System

An execution environment is necessary in order to be able to run the generated code. In PECOS, we have therefore defined the PECOS *Execution Environment* that abstracts from its underlying (Real-Time) OS and provides some language independent interfaces for synchronization. An execution environment for C++ and Java is defined that provides a common application programming interface (API) for both target languages.

The execution environment contains a highest-priority first, pre-emptive scheduler. Every active component (and its passive sub-components) in a CoCo specification is mapped to a separate thread in the target execution environment. The assignment of priorities, periods and deadlines for the tasks can be specified as timing properties for every active component. The actual schedule is computed from the values of these properties.

6. THE PECOS CBSE PROCESS

The PECOS project not only aimed at setting up the technical basis to enable component-based software development in the area of field device software but also delivered a software process model for component-based software development.

The PECOS process models the tasks and actions and their interrelation, that are typically performed over the course of a PECOS development. Since the PECOS process concentrates on a particular application domain, it has been adjusted to address the particular challenges of field device software as well as to incorporate the specific tools and methods to tackle them. This process covers all relevant development phases from requirements gathering to scheduler calculation, system simulation deployment and release.

The following sections give only an outline of the PECOS process. First, an overview on *application development* is given. Then, the tasks of *component composition* and *component development* are presented. A more

detailed description of the process can be found in [1]. The PECOS process is an iterative process. However, in the following section we often do not show iterations explicitly.

6.1 Application Development

Application development, as shown in Figure 10, is the process of producing a software system conforming to the *software requirements* of a PECOS field device. Besides the software part, a field device consists of two more parts: the mechanical and the hardware parts. As requirements specification only makes sense with respect to a common treatment of these three aspects, we do not investigate this any further. But we assume that the software, hardware and mechanical requirement specifications will be available.

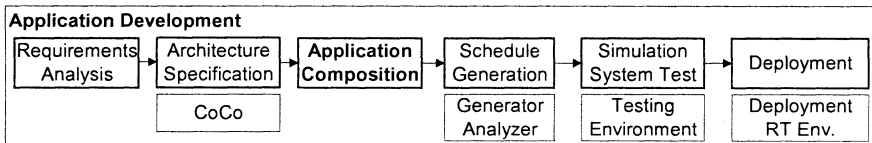


Figure 10. Application Development Tasks and Tools

Field devices are most probably part of a system family with similar requirements and architectures. This commands the adaptation and reuse of standard architectures rather than building everything from scratch. Therefore, the first step in every PECOS application development is to define an overall *system architecture* and implement it with the CoCo language, as outlined in Section 4.3 or to reuse and adapt an existing architecture. The use of standard architectures speeds up the whole development process, while using an approved and well-understood design at the same time.

The high-level application architecture of a system is the starting point for the actual composition of the application from components. This task, is also referred to as *application composition* (see Figure 11). It is organized as an iterative, incremental approach to refine the initial system decomposition until a fully specified device is built. The refinement process of every iteration applies the waterfall model and comes in guise of the four main tasks of *identifying*, *querying for*, *selecting/building* and *composing* components.

This process produces a series of partial, and finally a complete specification of the application. These are subject to the *schedule generation* task. During this activity, an application global schedule is produced, using the timing information associated with every component. It is supported by scheduling generation and analysis tools.

Finally, the application must be tested. Therefore, it is generally necessary to deploy the application onto the physical device, since many components are hardware dependent. This is especially necessary in order to test the generated schedule under real-world conditions.

6.2 Application Composition

The subsystem decomposition that results from the initial architecture is usually quite coarse-grained and can not be realized directly. More fine-grained components have to be *identified* that together realize these complex subsystems. Thus, incremental functional decomposition of the application has to be achieved.

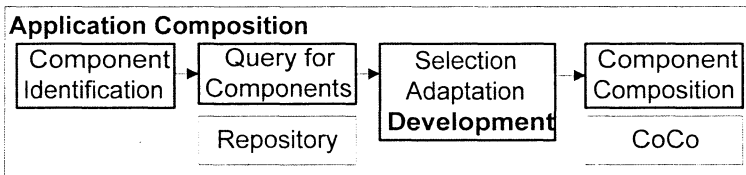


Figure 11. Application Composition Tasks and Tools

The *query for component* task is employed in order to find out about readily available components that can be used to realize a given system decomposition. This activity is supported by a *component repository*, which is used to store and recover components. It becomes clear, that identifying components (which is a top-down approach) and querying for components (which is a bottom-up approach) are actually highly coupled, as they influence each other mutually. Depending on the query results, different procedures are possible. In some cases it may be possible to directly reuse a component, in other cases an adaptation may be necessary. Yet, if no suitable components are available at all, new ones have to be developed. In this case, the sub-task of *component development* is triggered, which will be described in Section 6.4. Component composition is discussed in the following section.

6.3 Component Composition

The previous tasks are assumed to lead to a set of components, which are suitable to realize (parts of) the application. The next step to be taken is to compose them on the functional as well as on the non-functional level. This means that the *data-* and *control-flow* dependencies between them must be determined. The data dependencies are fixed by simply wiring the respective ports by connectors (see Section 4.1). The control-flow is determined by an

application-global schedule, which conforms to the timing requirements of the device.

Composition rules and contracts are used to specify constraints over a composition and therefore enable a correct-by construction approach. Rules designate constraints over a component or composition in terms of predicates over component properties as presented in Section 4.4. They only refer to statically available information and thus do not depend on information which is only available at run-time.

6.4 Component Development

Component development is a sub-task of application development and is triggered whenever components are required that can not be provided from the component repository directly or through adaptation of similar components.

Developing a PECOS component means to perform the following main tasks: *requirements elicitation*, *interface specification*, *implementation*, *testing & profiling*, *documentation* and finally *release* (see Figure 12). All of these activities are well-known in component based development, but they are special here, as they are tailored to the PECOS component model.

Requirements elicitation aims to define the functional as well as the *non-functional* properties of a component. The functional part concerns the interface comprising in- and out- ports as well as the functional mapping of in-port values to out-port values. The non-functional part primarily concerns properties covering timing information for the schedule generation, but also the type of the component (active, passive, event) and the target language.

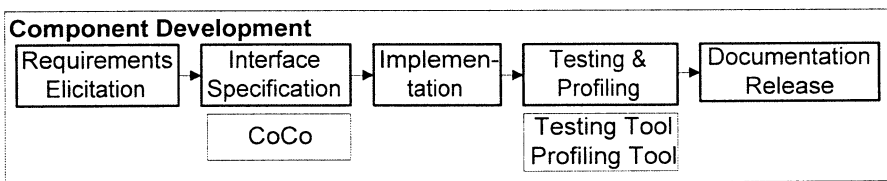


Figure 12. Component Development and Tools

Specifying the interface means writing down the component's characteristics in CoCo. It consists of its ports and properties identified during requirements elicitation. Implementing the actual behavior of the component requires to fill in its target language code into the predetermined hooks, as presented in Section 4.2. Testing is an important concern to ensure the software quality which is necessary in order to enable real reuse of components. Profiling is the task of determining a component's execution characteristics, like e.g. its worst case execution time, which is needed for

schedule generation. Documentation must be provided, if the component is to be released to the repository.

6.5 Tool Support

The PECOS process is supported by a number of different tools that help the developer to perform different actions. Among these tools are the CoCo language (as introduced in Section 4), a composition checker based on a Prolog system, standard repositories, such as integrated CVS version management support [18], editors, code browsers, code generators for the supported target languages, tools for schedule computation, the execution system etc.

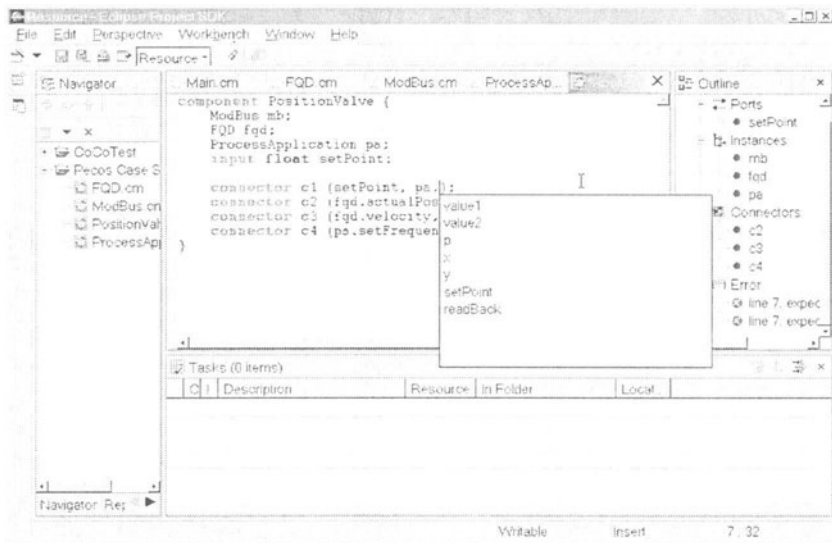


Figure 13. The PECOS Component Environment

Most of these tools are integrated in the PECOS Component Environment. This environment is an integrated development environment based on the ECLIPSE platform [19]. ECLIPSE is an open platform that provides the users of the PECOS technology with a professional basis for development tools. Due to the openness of the underlying ECLIPSE platform, the PECOS component environment can be extended with other tools such as graphical composition tools, simulators, remote debuggers, deployment tools, etc.

7. RELATED WORK

Several approaches for composing software applications out of readily available components have been proposed in the literature. The most significant contribution comes from the software architectures domain [6], [13]. Architecture systems introduce the notion of components, ports, and connectors as first class representations. However, most of the approaches proposed in the literature do not take the specific properties of embedded software systems into account.

In [20] the OCTOPUS approach for developing object-oriented software for embedded real-time systems is proposed. OCTOPUS is based on OMT and FUSION and is used in the telecommunication area. OCTOPUS defines a systematic development process for embedded software. OCTOPUS does, however, only provide limited architectural support. The support for non-functional aspects (e.g., memory consumption) of embedded software is also not apparent.

In [14] van Ommering *et al.* introduce a component model called Koala that is used for embedded software in consumer electronic devices. Koala components may have several *provides* and *requires* interfaces. Each interface defines *ports* which represent methods in the same way as under the object-oriented programming paradigm. In order to generate efficient code from Koala specifications, partial evaluation techniques are employed. However, Koala does not take into account non-functional requirements such as timing and memory consumption. Koala lacks a formal execution model and automated scheduler generation is not supported.

In [15] a framework for dynamically re-configurable real-time software is presented. It is based on the concept of so called Port Based Objects. The framework provides only a limited form of specifying a component (e.g., only rudimentary scheduling information is given, predefined port types). Furthermore the architecture is limited i.e., there is no support for composite components. The verification of a composition regarding non-functional properties such as memory consumption and schedulability is lacking, too.

8. CONCLUSION

Nowadays, software is becoming the major cost factor for embedded field devices. Current software development practice in this area is, however, still far away from what is standard in the information systems domain. Development from scratch is common practice and reuse and component-based software are foreign words.

In this chapter we presented the PECOS approach to the systematic construction of software for embedded field devices. The approach is based

on the component-based software development paradigm and has been adapted to the special requirements of the embedded world, as there are restricted resources (e.g., memory, low CPU performance), cyclic behavior, data-flow oriented interaction between parts of the system and threading.

As described in the paper, the PECOS approach consists of several ingredients. First of all there is a component model that represents components and the composition of these components together with an execution model that allows the reasoning about the behavior of the system. The language Coco is used to specify the components and compositions in a textual way. Coco can also be used to specify software families. We have shown, that Coco specifications can be translated into the general purpose programming languages C++ and Java. Finally we presented a process that describes how to derive an application out of requirements using the PECOS approach. The PECOS approach can help to reduce development costs for embedded systems and there are already several promising projects running at the PECOS industry partner site which use this approach.

ACKNOWLEDGEMENTS

The PECOS project has been funded by the European Commission under IST Program IST-1999-20398 and by the Swiss government as BBW 00.0170.

The work presented in this chapter is result of a joint effort of different people who worked together in PECOS. We would therefore like to thank all of these people for their major contributions and fruitful discussions. We particularly like to thank the following: Reinier van den Born and Bastiaan Schönhage (OTI, The Netherlands) for their work on the component model, the runtime system and the language mapping; Oscar Nierstrasz, Stephane Ducasse, Roel Wuyts (University of Berne) for their work on the component model and scheduling; Peter Müller and Christian Zeidler (ABB Research) and Andreas Stelter (ABB Automation) for their contribution to the PECOS CBSE process, the component model and for being the sanity checkers throughout the whole project.

REFERENCES

- [1] Michael Winter, Christian Zeidler, and Christian Stich, *The PECOS process*, ICSR7 2002 Workshop on component-based software development process, Austin, Texas.
- [2] Alan Burns and Andy Wellings, *Real-Time Systems and Programming Languages*, Addison Wesley, 1989.
- [3] PROFIBUS International, *PA General Requirements, Version 3.0*, www.profibus.org.
- [4] PROFIBUS International, *Device Datasheet for Actuators, Version 3.0*, www.profibus.org.

- [5] *Fast Quadrature Decode TPU Function (FQD)*, Semiconductor Motorola Programming Note, TPUPN02/D.
- [6] M. Shaw and D. Garlan, *Software Architecture -- Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
- [7] Jiacun Wang, *Timed Petri Nets*, Kluwer Academic Publishers, 1998.
- [8] O. Nierstrasz, S. Ducasse, R. Wuyts, Gabriela Arèvalo, A. Black, P. Müller, C. Zeidler, T. Genssler, and R. van den Born, *A component model for field devices*, 2nd Conference on Component Deployment, 2002.
- [9] Stéphane Ducasse and Roel Wuyts (editors), Field-device component model. Technical Report Deliverable D2.2.8, PECOS, 2001, www.pecos-project.org.
- [10] P.O. Müller, C. Stich, and C. Zeidler, *Components @ Work: Component Technology for Embedded Systems*, Euromicro Workshop on Component-based Software Engineering, Warsaw, Poland, 2001.
- [11] T. Genssler, A. Christoph, R. van den Born, *The CoCo Language Description*, Technical Report Deliverable D2.2.5, PECOS, 2002, www.pecos-project.org.
- [13] Clements, Paul C., *A Survey of Architecture Description Languages*, Int. Workshop on Software Specification and Design, 1996.
- [14] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee, *The Koala Component Model for Consumer Electronics Software*, IEEE Computer, 2000.
- [15] David B. Stewart, Richard A. Volpe, and Pradeep K. Khosla, *Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects*, IEEE Transaction on Software Engineering, 1997.
- [16] Embedded C++ Homepage, <http://www.caravan.net/ec2plus/>, 2002.
- [17] T. Genssler, C. Zeidler, *Rule-driven component composition for embedded systems*, Proceedings of the 4th ICSE Workshop on Component-Based Software Engineering, 2000.
- [18] CVS Homepage, <http://www.cvshome.org/>, 2002.
- [19] Eclipse Tool Platform, <http://www.eclipse.org/>, 2002.
- [20] M. Awad, J. Kuusela, J. Ziegler, *Object-Oriented Technology for Real-Time Systems*, Prentice Hall, 1996.
- [21] Bastiaan Schönhaage, *Model mapping to C++ or Java-based ultra-light environment*, Deliverable D2.2.9, www.pecos-project.org
- [22] *The ModBus web site*, www.modbus.org, 2002
- [23] L. Briand, L. Roy, *Meeting Deadlines in Hard Real-Time Systems – The Rate Monotonic Approach*, IEEE Computer Society, 1999

Chapter 11

Model-Based Risk Assessment in a Component-Based Software Engineering Process

The CORAS Approach to Identify Security Risks

Ketil Stølen¹, Folker den Braber¹, Theo Dimitrakos², Rune Fredriksen³, Bjørn Axel Gran³, Siv-Hilde Houmb⁴, Yannis C. Stamatiou⁵ and Jan Øyvind Aagedal¹

¹Sintef Telecom & Informatics, Norway; ²CLRC Rutherford Appleton Laboratory, UK; ³Institute for Energy Technology, Norway; ⁴Telenor R&D, Norway; ⁵Computer Technology Institute, Greece

Abstract: The EU-funded CORAS project (IST-2000-25031) is developing a framework for model-based risk assessment of security-critical systems. This framework is characterised by: (1) A careful integration of techniques and features from partly complementary risk assessment methods. (2) Patterns and methodology for UML oriented modelling targeting the different risk assessment methods. (3) A risk management process based on AS/NZS 4360. (4) A risk documentation framework based on RM-ODP. (5) An integrated risk management and system development process based on UP. (6) A platform for tool-inclusion based on XML. This chapter describes and explains the CORAS approach to model-based risk assessment. The ability to aid risk assessment in a component-based software engineering process receives particular attention. We consider maintenance, composition as well as reuse of risk assessment results.

Key words: Risk assessment, component-based software, modelling, IT security, maintenance

1. INTRODUCTION

CORAS [10] aims for improved methodology and computerised support for precise, unambiguous, and efficient risk assessment of security-critical systems. The CORAS project focuses on the tight integration of viewpoint-oriented semiformal modelling in the risk assessment process, in the

following referred to as model-based risk assessment. Model-based risk assessment differs from traditional risk assessment in the sense that it:

- combines complementary risk assessment methods for assessing different models of the target of evaluation;
- gives detailed recommendations for the use of modelling methodology in conjunction with risk assessment;
- provides modelling methodology to support the documentation of risk assessment results.

An important aspect of the CORAS project is the practical use of the Unified Modelling Language (UML) [32] and the Unified Process (UP) [21] in the context of security and risk assessment.

CORAS addresses security-critical systems in general, but places particular emphasis on IT security. IT security includes all aspects related to defining, achieving, and maintaining confidentiality, integrity, availability, non-repudiation, accountability, authenticity, and reliability of IT systems [17]. An IT system for CORAS is not just technology, but also the humans interacting with the technology and all relevant aspects of the surrounding organisation and society.

The CORAS consortium consists of three commercial companies: Intracom (Greece), Solinet (Germany) and Telenor (Norway); seven research institutes: CTI (Greece), FORTH (Greece), IFE (Norway), NCT (Norway), NR (Norway), RAL (UK) and Sintef (Norway); as well as one university college: QMUL (UK). Telenor and Sintef are responsible for the administrative and scientific coordination, respectively.

The remainder of the chapter is divided into six sections: Section 2 presents the CORAS framework. Section 3 motivates a contract-oriented approach to documenting risk assessment results. Sections 4, 5 and 6 consider maintenance, composition and reuse of risk assessment results, respectively. Section 7 provides a brief summary and the main conclusions.

2. THE CORAS FRAMEWORK

As illustrated in Figure 1, the main focus of the CORAS framework is model-based risk assessment; moreover, the framework is founded on four pillars: (1) A risk documentation framework based on RM-ODP [16]. (2) A risk management process based on AS/NZS 4360 [1]. (3) An integrated risk management and development process based on UP [21]. (4) A platform for tool-inclusion based on XML [5].

In the following subsections we describe the rationale behind the CORAS framework, its main focus as well as the four pillars.

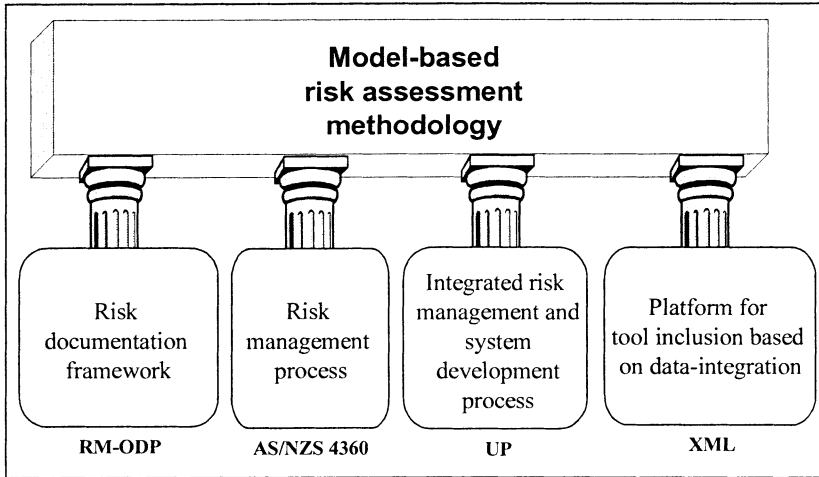


Figure 1. The CORAS framework

2.1 The rationale

As illustrated in Figure 2, model-based risk assessment employs modelling methodology for three main purposes: (1) To describe the target of evaluation at the right level of abstraction. (2) As a medium for communication and interaction between different groups of stakeholders involved in a risk assessment. (3) To document risk assessment results and the assumptions on which these results depend.

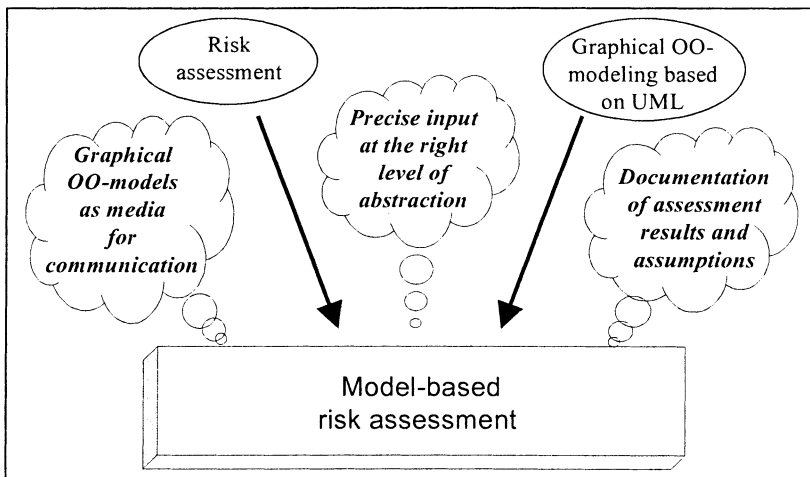


Figure 2. Model-based risk assessment

The choice of model-based risk assessment is motivated by several hypotheses:

- Risk assessment benefits from correct descriptions of the target of evaluation, its context and security issues. The modelling methodology furthers the precision of such descriptions, and this is likely to improve the quality of risk assessment results.
- The graphical style of UML facilitates communication and interaction between stakeholders involved in a risk assessment. This may improve the quality of risk assessment results, and reduce the danger of throwing away time and resources on misconceptions.
- The modelling methodology facilitates a more precise documentation of risk assessment results and the assumptions on which their validity depends. This is likely to reduce maintenance costs by increasing the possibilities for reusing and updating assessment results when the target of evaluation is maintained.
- The modelling methodology provides a solid basis for the integration of assessment methods. This may improve the effectiveness of the assessment process.
- The modelling methodology is supported by a rich set of tools from which the risk assessment benefits. This may improve the quality of assessment results and reduce costs. It may also further productivity and maintenance.
- The modelling methodology provides a basis for tighter integration of risk management in the system development process. This may considerably reduce development costs and ensure that the specified security level is achieved.

2.2 The risk documentation framework

The CORAS risk documentation framework is a specialisation of the Reference Model for Open Distributed Processing (RM-ODP) [16]. RM-ODP is an international standard reference model for distributed systems architecture, based on object-oriented techniques. RM-ODP divides the system documentation into five viewpoints. It also provides modelling, specification and structuring terminology, a conformance module addressing implementation and consistency requirements, as well as a distribution module defining transparencies and functions required to realise these transparencies.

The CORAS risk documentation framework is a specialisation of RM-ODP and can be understood as a reference framework for model-based risk assessment. RM-ODP contains many features that are not directly relevant for risk assessment. All RM-ODP features are, however, relevant for distributed systems. Since most IT systems of today are distributed or at least components of distributed systems, the CORAS risk documentation

framework contains RM-ODP in full. On the other hand, the CORAS risk documentation framework refines only those parts of RM-ODP that are directly relevant for risk assessment of security critical systems. The CORAS risk documentation framework refines RM-ODP in the following manner.

- The RM-ODP terminology is extended with two new classes of terminology, namely, concepts for risk assessment and security. Figure 3 illustrates the relationship between some of the most important notions in the risk assessment terminology.
- The RM-ODP viewpoint structure is divided into concerns targeting security in general and model-based risk assessment in particular. As illustrated in Figure 4, concerns may be understood as specialised cross-viewpoint perspectives linking together related information within the five viewpoints. The concerns are further decomposed into models. A model provides the content of a concern with respect to a particular viewpoint. For each model there are guidelines for its development, including concrete recommendations with respect to which modelling languages to use.
- The RM-ODP conformance module is extended with additional support for conformance checking targeting concerns.

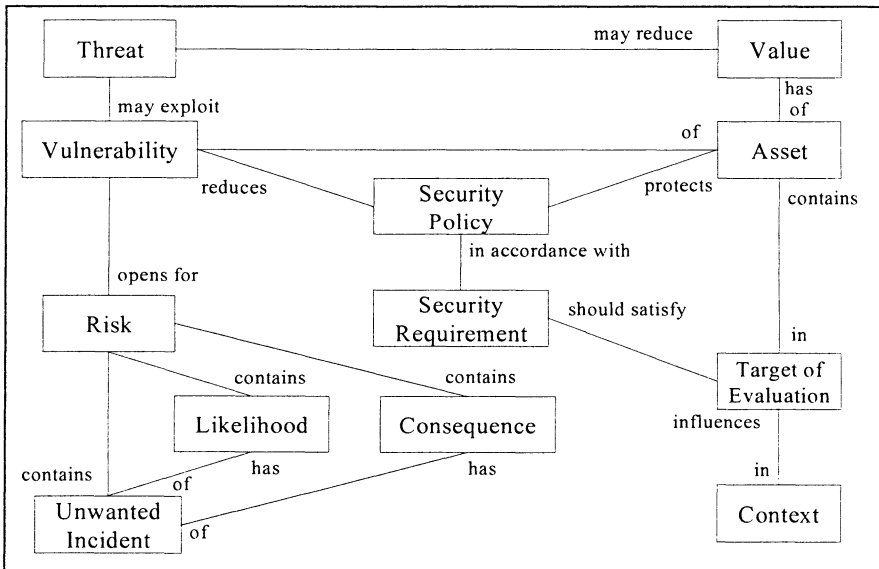


Figure 3. The CORAS terminology

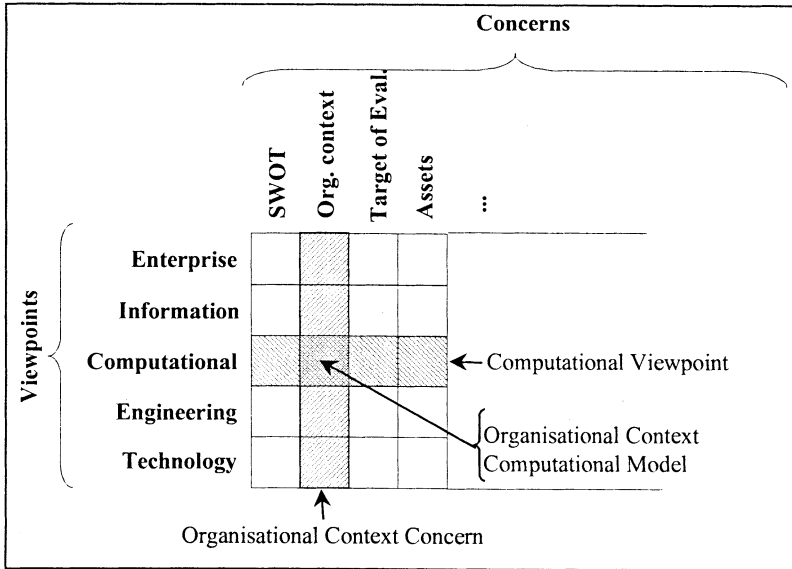


Figure 4. Relationship between viewpoints, concerns and models

The CORAS risk documentation framework also provides libraries of reusable elements. These may be understood as specification fragments or patterns and templates for formalising risk assessment results capturing generic aspects suitable for reuse.

Finally, there are also plans to extend RM-ODP with a specialised risk assessment module containing a risk assessment process, risk assessment methodologies, international standards on which CORAS builds as well as integration formats for computerised tools.

2.3 The risk management process

The CORAS risk management process is based on AS/NZS 4360:1999 Risk Management [1] and ISO/IEC 17799:2000 Code of Practice for Information Security Management [19]. Moreover, it is complemented by ISO/IEC 13335:2001 Guidelines for the management of IT-Security [17] and IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety Related Systems [15]. As illustrated in Figure 5, AS/NZS 4360 provides a sequencing of the core part of the risk management process into sub-processes for context identification, risks identification, risks analysis, risks evaluation, and risks treatment.

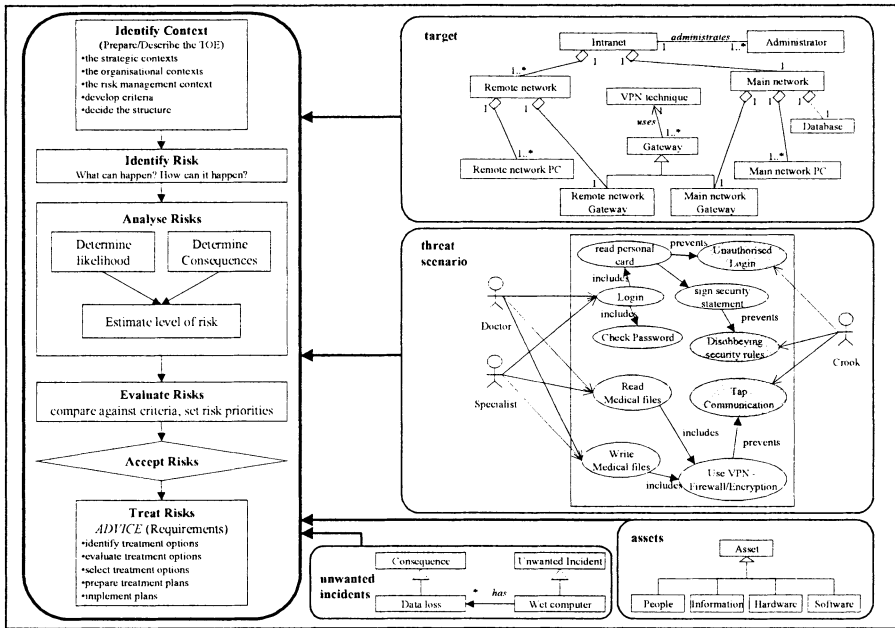


Figure 5. The role of UML in the CORAS risk management process

For each of these sub-processes, the CORAS methodology gives detailed advice with respect to which models should be constructed, and how they should be expressed. Figure 6 assigns concerns to the five sub-processes. Note that, even if the sub-processes are sequenced, AS/NZS 4360 is iterative and allows feedback.

Models expressed in the Unified Modelling Language (UML) [32] are used to support and guide the risk management process. The four diagrams to the right in Figure 5 illustrate:

- specification of the target of evaluation with the help of a UML class diagram (aspect of the *target of evaluation concern* listed in Figure 6);
- specification of a threat scenario with the help of a misuse case diagram [31] (example element of the *threat scenarios concern* listed in Figure 6);
- specification of the assets to be protected with the help of a UML class diagram (aspect of the *assets concern* listed in Figure 6);
- specification of an unwanted incident with the help of a UML class diagram (example element of the *unwanted incidents concern* in Figure 6).

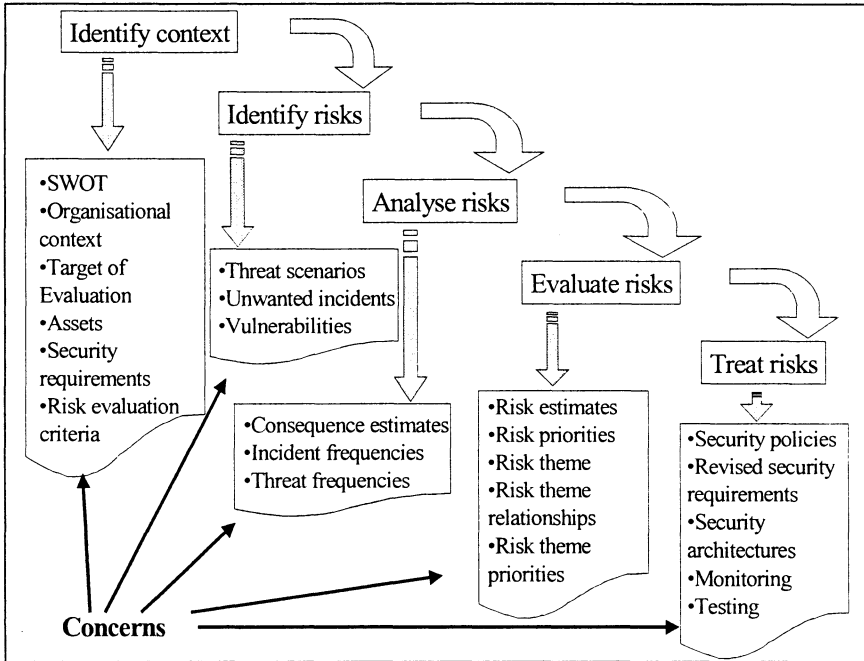


Figure 6. The relationship between concerns and the risk management process

2.4 The integrated risk management and system development process

The CORAS integrated risk management and system development process is based on an integration of the risk management process described above in the Unified Process (UP) [21]. In the following paragraphs we highlight the defining characteristics of this integrated process, as summarised in Figure 7.

In analogy to UP, the system development process is both stepwise incremental and iterative. In each phase of the system lifecycle, sufficiently refined models of the system are constructed through subsequent iterations. Then the system lifecycle moves from one phase into another. In analogy to the RM-ODP viewpoints, the viewpoints of the CORAS framework are not layered; they are different abstractions of the same system focusing on different groups of stakeholders. Therefore, information in all viewpoints may be relevant to all phases of the lifecycle.

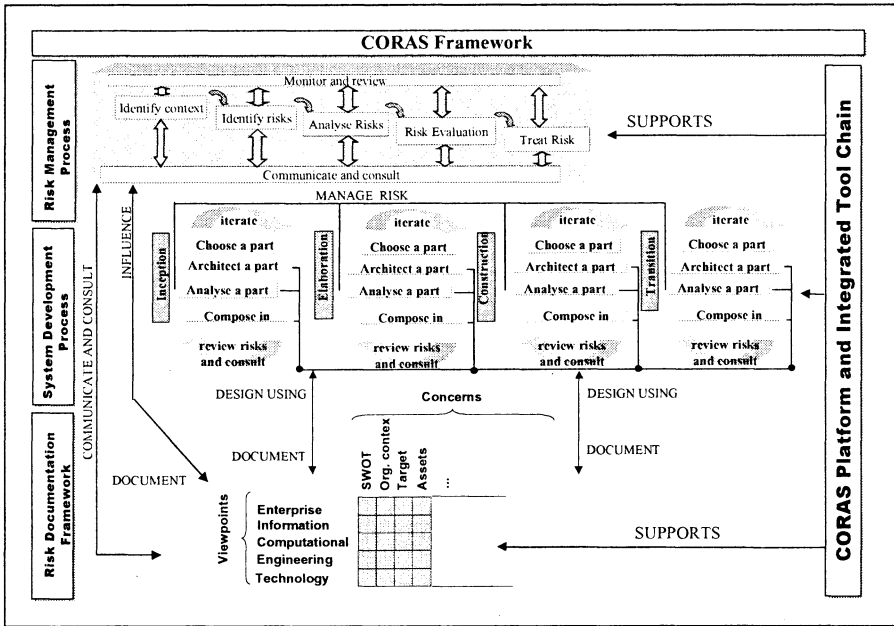


Figure 7. The integrated risk management and system development process

The risk management process follows the main iterations made in the system development process, as indicated in Figure 8. Each of the main iterations adds more detail to the target and the context of the assessment and previous results may need to be re-evaluated.

A set of agreed system requirements is one important outcome of the inception and elaboration phases. These requirements may be relevant to several viewpoints and can be described using a selection of different description methods, which are classified per concern. As one cannot expect that all security requirements are present from start, they have to be elicited. We anticipate that (appropriately adapted) model-based security risk assessment can also help with eliciting security requirements. However, risk assessment methods are traditionally designed to cope with unwanted incidents arising from design errors rather than specification problems related to missing requirements. For risk assessment to play a significant role in the elaboration phase, the CORAS risk assessment methods are being adapted to address requirement elicitation properly.

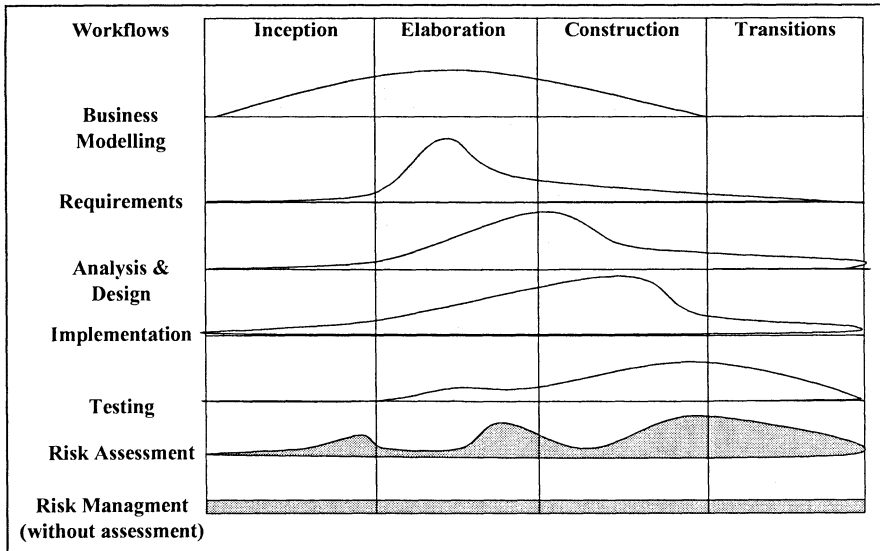


Figure 8. Relating risk management to system development

2.5 The platform for tool inclusion

A platform for tool inclusion based on data integration is under construction. Its internal data representation is formalised in the Extensible Markup Language (XML) [5]. Based on the Extensible Stylesheet Language Transformations (XSLT) [7], relevant aspects of this data representation may be mapped to the internal data representations of other tools (and the other way around). This allows the inclusion of sophisticated case-tools targeting system development as well as risk assessment tools and tools for vulnerability and threat management.

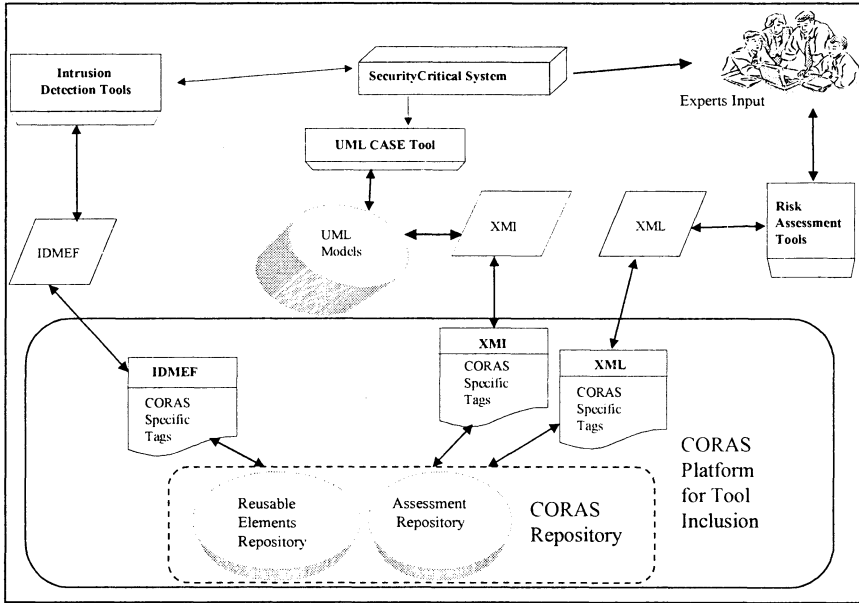


Figure 9. The platform for tool inclusion

As indicated in Figure 9, the CORAS platform is supposed to offer three interfaces for XML based data exchange:

- Interface based on the Intrusion Detection Exchange Format (IDMEF) [11]. IDMEF is an XML DTD targeting tools for intrusion detection and has been developed by the Intrusion Detection Working Group.
- Interface based on the XML Metadata Interchange (XMI) [29] standardised by the Object Management Group and targeting tools for UML modelling.
- Interface targeting risk assessment tools.

The CORAS platform will contain a repository divided into two sub-repositories: (1) The assessment repository storing the concrete results from already completed assessments and assessments in progress. (2) The reusable elements repository storing reusable models, patterns and templates from already completed risk assessments. Both sub-repositories mirror the decomposition into viewpoints and concerns illustrated in Figure 4.

2.6 The risk assessment methodology

The CORAS risk assessment methodology is a careful integration of techniques and formats inspired by HazOp Analysis [30], Fault Tree Analysis (FTA) [14], Failure Mode and Effect Criticality Analysis (FMECA) [4], Markov Analysis [25] as well as CRAMM [2].

The integrated risk assessment methods are to a large extent complementary. They address confidentiality, integrity, availability as well as accountability; in fact, as indicated by Table 1, all types of risks/threats/hazards associated with the target system can potentially be revealed and dealt with using these methodologies. They also cover all phases in the system development and maintenance process.

Table 1. The relevance of risk assessment methodologies

	HAZOP	FTA	FMECA	Markov	CRAMM
Identify context	In case of brief system description				Valuation of assets
Identify risks	Address all security aspects	Top-down starting from unwanted outcomes	Bottom-up for critical sub-parts		Focus on data groups
Analyse risks	As input for FTA/FMECA/Markov	Address top events, basic events, and likelihood	Address failure modes and consequences	Address system states, and likelihood	
Evaluate risks	As input	Compare with criteria	Compare with criteria	Compare with criteria	
Treat risks	Identify treatment options	Address priorities	Address barriers and counter-measures	Support maintenance	Identify counter-measures

3. CONTRACT-ORIENTED DOCUMENTATION OF ASSESSMENT RESULTS

Risk assessments are both costly and time-consuming, and cannot be carried out from scratch each time a system is updated or modified. This motivates the need for specific methodology addressing the maintenance of risk assessment results in particular, and a component-based approach to risk assessment in general.

In the following we propose an approach to component-based risk assessment based on contract-oriented documentation of risk assessment results. An assessment contract consists of two parts, an assessment assumption describing the target of evaluation as well as other pre-conditions on which the assessment builds, and an assessment guarantee describing

assessment results for the component in question with respect to the assessment assumption.

The documentation of risk assessment results in the form of assessment contracts, mirrors the contract-like flavour of the risk management process. As illustrated in Figure 5 the risk management sub-process “Identify Context” involves: (1) Establishing the strategic, organisational and risk management context. (2) Identifying and valuing assets. (3) Identifying existing security policies and formulating risk evaluation criteria. The concerns documenting the results from this sub-process constitute the assessment assumption.

The four subsequent sub-processes identifies, analyses, evaluates and treats risks with respect to the assessment assumption resulting from the “Identify Context” sub-process. In this sense, the concerns documenting the results from these four sub-processes constitute the assessment guarantee. Hence, with respect to Figure 6, the concerns listed under “Identify Context” records the assessment assumption, while the others capture the assessment guarantee.

In the following we outline how the CORAS approach may be used to support component-based risk assessment given that risk assessments are documented in the form of assessment contracts as suggested above.

4. MAINTAINING ASSESSMENT RESULTS

IT systems are updated or modified on a regular basis. Connected to such updates or modifications it is often necessary to reassess their security since changes may have introduced new risks. In the next two sub-sections we consider maintenance of assessment results with respect to two different kinds of component modifications.

4.1 When components are maintained

In the following we address maintenance of risk assessment results for the situation where a component for which we have already carried out a risk assessment is updated or adjusted without being changed in a fundamental manner.

The target for a risk assessment may only be a certain part or feature of the component in question. Hence, the first step when maintaining a risk assessment for a component undergoing minor updates is to check whether the updates and adjustments are within the target of evaluation. If they are not and they do not invalidate the conditions put down in the assessment assumption, the existing assessment carry over unchanged.

On the other hand, if the updates and adjustments are within the target of evaluation or invalidate the assessment assumption, it will be necessary to

reassess at least some of the concerns documenting the assessment guarantee. CORAS is developing specialised rules and guidelines to support this kind of reassessment exploiting relationships and dependencies between concerns.

4.2 When components are replaced

In the following we address the situation illustrated in Figure 9, where a component for which we have already carried out a risk assessment is updated by replacing one of its sub-components by a new sub-component. Assume we have a component $A+B+C$ consisting of three sub-components A, B and C. Moreover, assume we have carried out a risk assessment for $A+B+C$ documented by the assessment $RA:A+B+C$. If sub-component C is replaced by sub-component D then we would like a strategy for making use of the assessment $RA:A+B+C$ to arrive at an assessment $RA:A+B+D$ of the new component $A+B+D$.

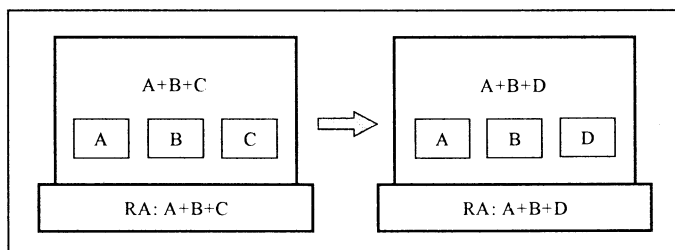


Figure 9. Replacing sub-component C by D

In accordance with the previous case, the first step is to situate the old and the new component with respect to the target of evaluation and the other conditions put down in the assessment assumption. If neither C nor D are situated within the target of evaluation, and D does not invalidate the assumptions on which the assessment $RA:A+B+C$ depends, the validity of the existing assessment carries over to $A+B+D$.

If this is not the case, specialised rules and guidelines exploiting relationships and dependencies between concerns may be used to simplify the reassessment. One simple rule is, for example, to check whether C contains a security asset or not. If it does not, and in addition, black-box testing implies that any external behaviour of D is an external behaviour of C, and D is without security assets, we may conclude that the assessment results for $A+B+C$ remains valid for $A+B+D$.

Black-box testing will of course normally not cover all cases (there will typically be infinitely many); hence, there is no guarantee that an important test-case is not left-out. On the other hand, there is no guarantee that a risk assessment will discover all threats.

5. COMPOSING ASSESSMENT RESULTS

In the following we address the situation illustrated in Figure 10, where two components, A and B, for which we have risk assessment results RA:A and RA:B, respectively, are composed. We indicate ways in which the CORAS approach may support the deduction of risk assessment results for the composite component from the assessment results for A and B.

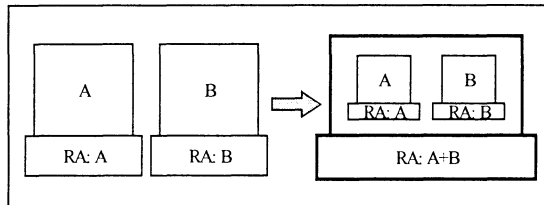


Figure 10. Composing assessment results

The first step is to situate the components A and B with respect to the assumptions of the assessments RA:B and RA:A, respectively. If we from this inspection conclude that the component A and its composition with B does not affect the target of evaluation RA:B nor any of the additional conditions put down in its assessment assumption, and accordingly for B with respect to RA:A, it follows that the assessments RA:A and RA:B are valid for A+B (note that the assumptions of the two assessments remain unchanged).

The premises for this inference will of course often be invalid, in which case more sophisticated rules and guidelines will be required to make full use of the already existing assessment results. Rules and guidelines of this kind are under development in the CORAS project.

6. REUSING ASSESSMENT RESULTS

Traditionally, system development methodologies focus on the development of single systems. More recently, the emphasis has shifted towards the development of system product lines. Inspired by [9], we define a system product line, as a set of systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. According to [2], components provide the perfect foundation for the practical application of product line development. Other examples of product-line oriented system development methods are FODA [23], FAST [33] and TIME [6]. In the following we outline CORAS support for reuse of assessment results in a product-line oriented system development.

A critical factor in the success of a product line approach is the nature of the reusable “core”. As argued in [2], “... at a minimum this should contain a reference architecture supported by techniques for capturing and selecting the points of variation among family members. In its most general form, the reusable asset takes the form of a framework, which embodies implementation (i.e., code level) artefacts for the common parts of the family, as well as higher-level design and architecture models. Since they embody every possible reusable asset, at all levels of abstraction, frameworks constitute the largest possible reusable artefacts for a particular product family.”

The CORAS framework is intended to support a product-line approach to system development in the following sense.

- For each artefact we may store assessment relevant information in the reusable elements repository. When a development of a new product is initiated, this assessment relevant information will be loaded into the assessment repository for each artefact to be reused.
- If the particular use of an artefact requires maintenance, the assessment relevant information will first be updated based on the strategy outlined in Section 4.1.
- If a new product requires replacing a sub-component in an artefact to be reused by another sub-component, the assessment relevant information will first be updated based on the strategy outlined in Section 4.2.
- To the extent a new product requires composing artefacts, an assessment of the composite artefact may be carried out based on the strategy outlined in Section 5.
- To the extent a new product also requires the development of completely new components from scratch, the reassessment of early assessment results (for example, the results from an assessment carried out during the inception phase) at a later point in the development may benefit from the strategies outlined in Sections 4.1, 4.2 and 5.

7. CONCLUSIONS

CORAS advocates model-based risk assessment. Model-based risk assessment is put forward as a means to improved efficiency of the risk assessment process as well as more reliable assessment results, since the understanding of the target of evaluation is enhanced by precise specifications of its structure and behaviour. Firstly, we argue that UML diagrams give a superior specification of system behaviour compared to free text or other informal formats. Secondly, a model-based risk assessment facilitates communication, both internally between the actors involved in the risk assessment and externally to the stakeholders. Thirdly, improved precision is

not only of importance to understand the target of evaluation and the set of possible threats, but also for the documentation of the risk assessment results and the assumptions on which their validity depends. As explained in Sections 3-6, structured documentation of risk assessment results and the assumptions on which they depend provides the basis for maintenance of assessment results as well as a component-based approach to risk assessment.

The development of the CORAS methodology and framework is guided by concrete experiences from two major trials, one within e-commerce and one within telemedicine. Both trials are divided into three trial sessions.

There are of course other approaches to model-based risk assessment. See for instance CRAMM [3], ATAM [8], SA [34] and RSDS [24]. The particular angle of the CORAS approach with its emphasis on security and risk assessment tightly integrated in a UML and RM-ODP is however new.

Contract-oriented specification has been suggested in many contexts and under different names. Within the RM-ODP community one speaks of contracts related to quality of service specification [12]. In the formal methods community there are numerous variations; the pre/post [13], the rely/guarantee [22] and the assumption/guarantee [28] styles are all instances of contract-oriented specification. Other more applied examples are the design-by-contract paradigm, introduced by Bertrand Meyer [26], and the UML based approach advocated by Mingins/Liu [27].

Since 1990, work has been going on to align and develop existing national and international schemes in one, mutually accepted framework for testing IT security functionality. The Common Criteria (CC) [18] represents the outcome of this work. The Common Criteria project harmonises the European “Information Technology Security Evaluation Criteria (ITSEC) [20]”, the “Canadian Trusted Computer Product Evaluation Criteria (CTCPEC)” and the American “Trusted Computer System Evaluation Criteria (TCSEC) and the Federal Criteria (FC)”. The CC is generic and does not provide methodology for risk assessment. CORAS, on the other hand, is devoted to methodology for risk assessment. Both the CC and CORAS places emphasis on semiformal and formal specification. However, contrary to the CC, CORAS addresses and develops concrete specification technology addressing risk assessment. The CC and CORAS are orthogonal approaches. The CC provides a common set of requirements for the security functions of IT products and systems, as well as a common set of requirements for assurance measures applied to the IT functions of IT products and systems during a security evaluation. CORAS provides specific methodology for one particular kind of assurance measure, namely risk assessment of security critical systems.

REFERENCES

- [1] AS/NZS 4360:1999 Risk management.
- [2] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., Zettel, J. Component-based product line engineering with UML. Addison-Wesley, 2002.
- [3] Barber, B., Davey, J. The use of the CCTA risk analysis and management methodology CRAMM. Proc. MEDINFO92, North Holland, 1589–1593, 1992.
- [4] Bouti, A., Ait Kadi, D. A state-of-the-art review of FMEA/FMECA. International Journal of reliability, quality and safety engineering 1:515-543, 1994.
- [5] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E. Extensible markup language (XML) 1.0 (Second edition). World Wide Web Consortium recommendation REC-xml, October 2000.
- [6] Bræk, R., Gorman, J., Haugen, Ø., Melby, G., Møller-Pedersen, B., Sanders, R. Quality by construction exemplified by TIME - the integrated methodology. Teletronikk 95(1):73-82, 1999.
- [7] Clark, J. XSL transformations (XSLT) 1.0, World Wide Web Consortium recommendation REC-xslt, November 1999.
- [8] Clements, P., Kazman, R., Klein, M. Evaluating software architectures: methods and case studies. Addison-Wesley, 2002.
- [9] Clements, P., Northrop, L. Software product lines: practices and patterns. Addison-Wesley, 2001.
- [10] CORAS: A platform for risk analysis of security critical systems. IST-2000-25031, 2000. (<http://www.nr.no/coras/>)
- [11] Curry, D., Debar Merrill Lynch, H. Intrusion detection message exchange format (IDMEF). Working draft, December 28, 2001.
- [12] Fevrier, A., Najm, E., Stefani, J. B. Contracts for ODP. Proc. ARTS97, LNCS, 1997.
- [13] Hoare, C. A. R. An axiomatic basis for computer programming. Communications of the ACM, 12:576-583, 1969.
- [14] IEC 1025: 1990 Fault tree analysis (FTA).
- [15] IEC 61508: 2000 Functional safety of electrical/electronic/programmable safety related systems.
- [16] ISO/IEC 10746: 1995 Basic reference model for open distributed processing.
- [17] ISO/IEC TR 13335-1:2001: Information technology – Guidelines for the management of IT Security – Part 1: Concepts and models for IT Security.
- [18] ISO/IEC 15408:1999 Information technology – Security techniques – Evaluation criteria for IT security.
- [19] ISO/IEC 17799: 2000 Information technology – Code of practise for information security management.
- [20] Information technology security evaluation criteria (ITSEC), version 1.2, Office for Official Publications of the European Communities, June 1991.
- [21] Jacobson, I., Rumbaugh, J., Booch, G. The unified software development process. Addison-Wesley, 1999.
- [22] Jones, C. B. Development methods for computer programs including a notion of interference. PhD-thesis, Oxford University, 1981.
- [23] Kang, K. C., Cohen, S. G., Novak, W. E., Peterson, A. S. Feature-oriented domain analysis (FODA) feasibility study. Technical report UMIAC-TR-21, SEI, 1990.
- [24] Lano, K., Androutsopoulos, K., Clark, D. Structuring and design of reactive systems using RSDS and B. Proc. FASE 2000, LNCS 1783, 97-111, 2000.

- [25] Littlewood, B. A reliability model for systems with Markov structure. *Appl. Stat.* 24:172-177, 1975.
- [26] Meyer, B. *Object-oriented software construction*. Prentice Hall, 1997.
- [27] Mingis, C., Liu, Y. From UML to design by contract. *Journal of object-oriented programming*, April issue: 6-9, 2001.
- [28] Misra, J., Chandy, K. M. Proofs of networks of processes. *IEEE transactions on software engineering*, 7:417-426,1981.
- [29] OMG-XML Metadata Interchange (XMI) Specification, v1.2, <http://www.omg.org>.
- [30] Redmill, F., Chudleigh, M., Catmur, J. *Hazop and software hazop*. Wiley, 1999.
- [31] Sindre, G., Opdahl, A. L. Eliciting security requirements by misuse cases. In *Proc. TOOLS_PACIFIC 2000*. IEEE Computer Society Press, 120-131, 2000.
- [32] UML proposal to the Object management group, Version 1.4, 2000.
- [33] Weiss, D. M. and Lai, C. T. R. *Software product line engineering: a family based software engineering process*. Addison-Wesley, 1999.
- [34] Wyss, G. D., Craft, R. L., Funkhouser, D. R. *The use of object-oriented analysis methods in surety analysis*. SAND Report 99-1242. Sandia National Laboratories, 1999.

Chapter 12

A Vocabulary of Building Elements for Real-Time Systems Architectures

Jose Luis Fernández-Sánchez

Universidad Politécnica de Madrid, Spain

Abstract: This chapter describes the elements that define the PPOOA vocabulary created for its application in Real-Time Systems architecture development. These building elements are: components, or computation entities, and coordination mechanisms, responsible for data transmission and process synchronization. Each of these elements has several real-time attributes that allow a design assessment using analytical methods. In this case, we recommend the “Rate Monotonic Analysis” time responsiveness assessment techniques. PPOOA is described here as a UML Profile, that is a group of UML model elements customized for real-time systems domain.

Key words: Real-Time systems, UML, component-based development, software architecture

1. INTRODUCTION

A software architectural style defines a family of systems in terms of structural and organizational patterns. This means that a software architectural style is defined by a vocabulary of building elements that can be used by developers considering the constraints and design rules imposed by the architecture style.

A comparison can be made with building styles; the building elements (arcs, columns, etc.) and their organization makes a gothic cathedral different from a romanic cathedral in spite of being used for the same purpose: worship.

PPOOA, Pipelines of Processes in Object Oriented Architectures, is a software architecture style to be used in those object oriented systems in which individual paths of execution are required to be concurrent and multiple processes may be positioned along the path to control the action [1].

This chapter presents the PPOOA vocabulary or classification of the building elements: components and coordination mechanisms, that will be used in the real-time applications development, using the above mentioned software architecture style.

Section 2 describes other related vocabularies. Section 3 describes the PPOOA component and coordination mechanism concepts. Section 4 presents the catalogue of PPOOA components. Section 5 describes PPOOA coordination mechanisms features. Section 6 outlines UML metamodel extension to include PPOOA building elements.

2. RELATED WORK ON REAL-TIME SYSTEMS VOCABULARIES

There are other vocabularies for real-time systems that have been developed before this one and that have been used as a main reference.

Frankel describes a vocabulary for building elements in Ada 83, and therefore building elements are considered as Ada packages. These are Definition Package, Abstract Data Type, Flow Package, Algorithmic Package, Entity Package and Active Class Package [2].

Jeffay proposes a specific vocabulary for real-time systems supported by a theory that analyzes real-time properties. This theory permits the scheduling of the system taking into account the frequency of tasks, the entry flows in the external devices and the load placed on shared resources. The applications considered are multimedia applications. The author proposes the following vocabulary: Devices, Processes, Resources and Coordination Mechanisms [3].

Burns et al, supply a vocabulary and a methodology in order to cover the deficiencies found in the HOOD (Hierarchical Object Oriented Design) methodology used by the ESA (European Space Agency) in the development of space systems. The central deficiencies were the support of periodic and aperiodic processes and the coordination of concurrent flows of activities. The authors of HRT-HOOD propose a vocabulary including passive object, active object, protected object, cyclic object and sporadic object [4].

The vocabulary is supported by a scheduling theory. The vocabulary proposed in HRT-HOOD is determined by implementation aspects and this is why it is useful for detailed design. It would not be easy to apply it in preliminary designs, principal goal of the PPOOA vocabulary, or in the description of a reference architecture. However, it has been very useful for the construction of the vocabulary proposed in the PPOOA style.

Other vocabularies have recently been proposed in conjunction with UML standard evolution.

Gomaa provides a vocabulary that organizes classes into groups considering that different types of systems will have a preponderance of classes in one or another category. The object structuring categories are as follows: Interface object, Entity object, Control object and Application logic object. The vocabulary is complemented with a method for concurrent object modeling [5].

Another recent vocabulary for real-time systems is proposed by Rational. The central architectural entity is the capsule. Capsules may have one or more special attributes called ports. Ports perform a two-way interface function on behalf their capsule [6].

3. PPOOA VOCABULARY DESCRIPTION

The vocabulary of building elements for real-time systems proposed in the PPOOA style consists of: Component and Coordination Mechanism (Figure 1).

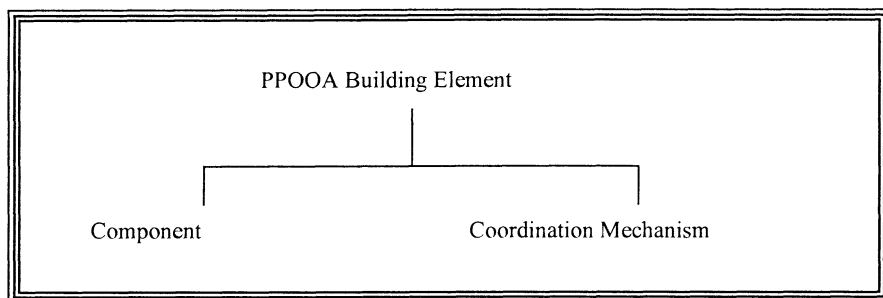


Figure 1. Basic Vocabulary of the PPOOA Architecture Style

Each of the basic elements is specialized in several kinds of elements as described later.

3.1 Components

The component represents a computing entity implementing one or more activities. It may provide and require interfaces to other components.

The PPOOA component is not exactly the same building element as the UML component, that is considered as a physical or replaceable part of a system.

PPOOA components are classified as: Algorithmic Component, Domain Component, Process, Structure, Controller Object and Subsystem.

Figure 2 shows the chosen vocabulary because it allows the representation of objects and explicit concurrency for an architecture in the real-time domain. The description of each component type is presented in the next section.

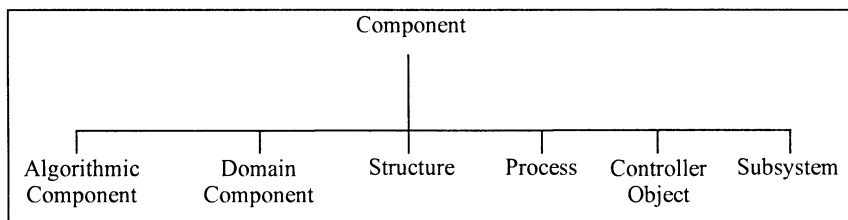


Figure 2. Architecture Components

3.2 Coordination Mechanisms

A coordination mechanism is an essential element of the PPOOA vocabulary. Its classification and categorization have been an essential part of PPOOA development.

A coordination mechanism provides the capabilities to synchronize or communicate components. Synchronization implies the blocking of a process until some specified condition is met. Communication is the transfer of information between components.

As part of the development of the PPOOA vocabulary, fourteen coordination mechanisms were analyzed and a taxonomy was developed [7]. Of those mechanisms, the following have been finally selected:

- Bounded Buffer
- General Semaphore
- Mailbox
- Transporter
- Rendezvous

These mechanisms have been selected because they are frequently used in pipelined real-time software architectures.

Figure 3 shows a scheme with the coordination mechanisms vocabulary. The description of these mechanisms is presented later.

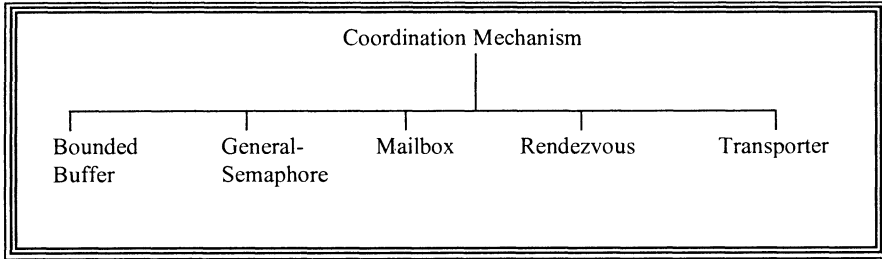


Figure 3. Coordination Mechanisms

4. PPOOA COMPONENTS DESCRIPTION

The proposed PPOOA components that are the building elements of the architecture are described in three ways: Concept, Context and Content of the component. In the description of a concrete system architecture, the detailed context and content or implementation issues, of each participant component will be provided.

The concept or supported abstraction is provided through a brief textual description. Also an enumeration of real-time attributes that describe the component behavior from a time responsiveness point of view, is given.

The provided operations are described with an architectural language, similar to UniCon [8] that emphasizes the structural aspects of the interface.

The notation used is:

INTERFACE IS	
NAME	operation name
TYPE	operation type (Reading, Writing, Computing, Signaling)
PARAMETERS	(parameters, types of parameters)

As a description enhancement, the Object Constraint Language (OCL) may be used to describe each interface operation precondition and postcondition [9].

Implementation issues are described. Particularly, design decisions that may impact on real-time behavior, and component composition constraints, that may impact on the structural properties of the developed architecture.

Information about the interfaces required by each component is also provided when a concrete architecture is designed.

An important issue is to take into account whether the component requires or not an execution flow (“thread”) independent of the other components of the architecture.

Several instances of the same component may participate in a concrete architecture for a concrete system.

4.1 Algorithmic Component

1. Abstraction supported

Algorithmic components or utilities are elements of the architecture that perform calculations or transform data from one type to another but are separated from its structural abstraction. They are typically represented by data classification components or data processing algorithms.

The algorithm component in PPOOA is not exactly the same as the utility defined in the UML metamodel since the algorithmic component may have instances.

Real-time attributes: execution time of the activities implemented by the algorithmic component.

2. Interface Provided

It basically provides computing operations by calling implemented methods.

3. Implementation

This component does not need an independent execution flow, since this component does not implement parallel activities.

If this component is an aggregate, it can only be decomposed into: algorithmic component, domain component or structure. However, this component is usually a primitive or atomic component.

4.2 Domain Component

1. Abstraction supported

The domain component is an element of the architecture that responds directly to the modeled problem. This component does not depend on any hardware or user interface. It resembles the class concept in oriented object design.

Real-time attributes:

- Execution time of each activity implemented by the domain component.
- Blocking. Blocking is a waiting condition that is not attributable to preemption. Blocking time estimation is done at architecture assessment.
- Priority (optional). Priority policy has to be considered if the concurrent access of the domain component is permitted.

2. Interface provided

It basically provides reading, writing or computing operations by calling implemented methods.

3. Implementation

This component does not need an independent execution flow. This component does not also implement parallel activities.

If this component is an aggregate, it can only be decomposed into: algorithmic component, domain component or structure. In addition, it is quite

usual that this component has persistence requirements. Finally, the blocking times, if any, have to be analyzable.

4.3 Structure

1. Abstraction supported

A structure is a component that denotes an object or class of objects characterized as an abstract state machine or an abstract data type.

Typical examples are: stack, queue, list, ring and others.

Real-time attributes: execution time of each activity implemented by the structure.

2. Interface provided

It basically provides reading or writing operations by calling implemented methods. These operations are also known as selectors or constructors.

A selector is an operation that evaluates the state of the structure. A constructor is an operation that alters the state of the structure. In the most general case, an iterator can also be supported. An iterator is an operation that allows visiting all of the parts of the structure.

3. Implementation

This component only allows concurrent operations when they are implemented in a protected way [4]. This means that the time when the calls to operations are executed is controlled and they cannot call spontaneously operations in other components. In general they cannot have arbitrary synchronous constraints and their blocking times have to be analyzable. They do not require an execution flow independent of the caller.

Structure is considered as a primitive or atomic component so it cannot be decomposed into other components.

4.4 Process

1. Abstraction supported

The process is a building element of the architecture that implements an activity or group of activities that can be executed at the same time as other processes. Its execution can be scheduled.

Real-time attributes:

- Execution time of each activity implemented by the process.
- Priority.
- Shared resources blocking. Blocking time estimation is done at architecture assessment.
- Offset (optional).

The PPOOA vocabulary supports two different types of processes: cyclic and aperiodic. Each of them is now described including their specific real-time attributes.

- a) Cyclic process: the cyclic process is used to implement an activity or group of activities that execute periodically. It has the following specific real-time attributes: execution period.
- b) Aperiodic process: the aperiodic process is used to implement an activity or group of activities that execute aperiodically. It has the following specific real-time attributes: activation pattern characterizing process activation occurrences.

2. Interface provided

In general, cyclic processes do not provide operations. The aperiodic process provides an activating operation in order to execute its executing flow. This operation should not block the caller. Other operations can be provided, but they should be executed immediately when called. They are called unconstrained operations in the HRT-HOOD terminology [4].

3. Implementation

- a) Cyclic process: in the proposed architectural style, the cyclic process communicates with other processes by means of coordination mechanisms, which are described later. The only operations that a cyclic process can supply are asynchronous control transfers, generally used to notify mode changes or error conditions. As it is an aggregate component, the cyclic process can include domain components, structures, cyclic processes and aperiodic processes. The constraints of the aggregation are imposed by temporal requirements.
- b) Aperiodic Process: the aperiodic process can supply other operations different from the activation operation, but these have to be unconstrained as mentioned above, and they have to be executed in the same way as the operations provided by domain components. The activation operation should not block the caller. As it is an aggregate component, the aperiodic process can include algorithmic components, domain components, structures, cyclic processes and aperiodic processes. The constraints of the aggregation are imposed by temporal requirements.

4.5 Controller Object

1. Abstraction supported

The controller object is responsible for initiating and managing directly a group of activities that can be repetitive, alternative or parallel. These activities can be executed depending on a number of events or circumstances.

Typically, a controller object receives an event. An event is something that can be notified to the system, like a POSIX signal, a hardware interrupt or a computed event, like an airplane entering a forbidden region or an alarm.

When one of these events occurs, the system schedules the associated event handlers.

A controller object manages two things: the dispatching of handlers when the event is fired, and the set of handlers associated with the event. The application can query the set and add or remove handlers.

The controller object has associated scheduling parameters that control the actual execution of the handler once it is fired. When an event is fired, the system executes the handlers asynchronously, scheduling them according to their parameters. The result is that the handler appears to have been assigned to its own thread. It is managing other components rather than supplying operations that depend on the application domain.

Real-time attributes:

- Priority per thread.
- Shared resources blocking. Blocking time estimation is done at architecture assessment.
- Execution time per activity implemented by the controller.
- Deadline or latest permissible completion time measured from the release time of the associated invocation of the schedulable object.

2. Interface provided

It supplies reading, writing, computing or signaling operations in a general way.

3. Implementation

The controller object is the most complicated component of all from an implementation point of view. Its implementation can be quite similar to that of the active object defined by HRT-HOOD [4]. The component can control the moment when the operation calls are executed and can make operation calls to other objects spontaneously. As the controller object is quite complicated, problems can arise in the schedulability analysis. This is why its use should be limited.

A possible implementation is the asynchronous event handling facility of real-time Java. It comprises two classes: `AsyncEvent` and `AsyncEventHandler`. An `AsyncEvent` manages two things: the dispatching of handlers when the event is fired, and the set of handlers associated with the event. An `AsyncEventHandler` is a schedulable object roughly similar to a Java thread [10].

In case the controller object is an aggregate component, it can include any type of component, even a component of the same type.

4.6 Subsystem

1. Abstraction supported

A subsystem is a component that clusters other components of the architecture based on the minimizing coupling and enhancing visibility criteria.

Subsystems have the following characteristics:

- Logic coherence. The provided operations are related to one another logically.
- Independence. Subsystems can be implemented as implementation components.
- Simple interfaces. Subsystems communicate with each other through simple and well defined interfaces.

The coupling reduction and the supply of simple interfaces are achieved in subsystems considering the “Façade” design pattern [11]. A façade supplies only one view of the subsystem that is useful for its clients. The Façade knows exactly what components of the subsystem are responsible for solving a certain operation request and delegates such request to the corresponding components.

The Façade pattern is not used when asynchronous communication or synchronization between process components, are established between processes allocated to different subsystems. In this case, coordination mechanisms are used as intermediaries in an architecture developed with the PPOOA style vocabulary.

2. Interface provided

The subsystem provides, through its Façade, general reading, writing, computing or signaling operations. As mentioned above, coordination mechanisms are used to synchronize and communicate process components contained in the subsystem.

3. Implementation

The subsystem supplies an interface that is implemented as a method invocation. As subsystems are building elements that cluster other elements, they may contain any other element of the PPOOA vocabulary.

5. PPOOA COORDINATION MECHANISMS

One of the contributions of the PPOOA vocabulary is the ability to model the coordination mechanisms properties accordingly to the FODA Feature-Oriented Domain Analysis methodology [12]. FODA was applied as a novelty because it was the first time that it was used in the analysis and classification taxonomy of fourteen real-time coordination mechanisms [7]. Not all of these mechanisms have been chosen as building elements in the PPOOA vocabulary.

The identification attributes represent those features that describe how coordinating partners refer to each other. There are two main categories or identification schemes: direct and indirect. Direct naming is used when the process component that needs to be synchronized or needs to communicate with another has to explicitly name the recipient partner. In contrast, indirect naming is used when the processes that need to be coordinated utilize some

type of intermediate object, the coordination mechanism, where they introduce or extract the information that is going to be transmitted, so that the identity of coordinating partners remains hidden. The synchronizing attributes describe those features related to the exclusion and ordering constraints of the coordination mechanisms (Figure 4).

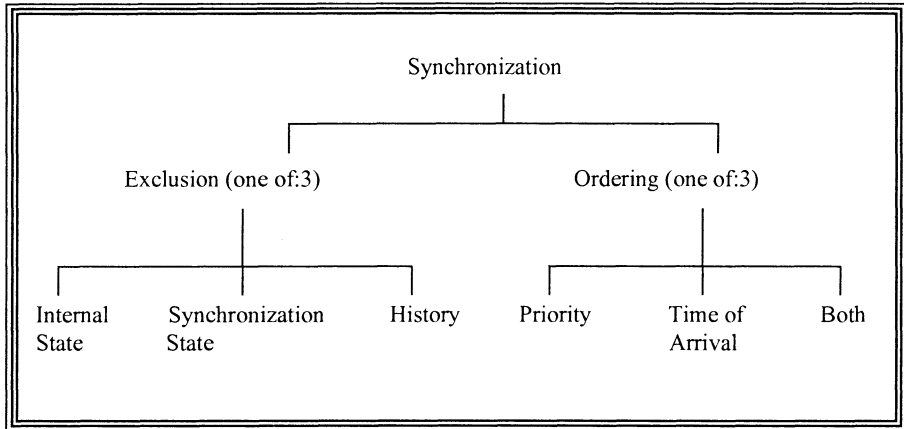


Figure 4. Synchronisation Features

The exclusion constraint features describe the properties that guarantee the blocking of certain processes that are attempting to access the coordination mechanism when they would interfere with the activities already in progress by other processes. Some coordination mechanisms support exclusion of client processes according to the internal state of the mechanism. A second group of mechanisms supports exclusion according to the synchronization state of the mechanism. A third group of mechanisms supports exclusion according to the history that has occurred previously. Exclusion also considers the number of client processes that are allowed to access the mechanism.

Ordering constraint features are concerned with the prioritization of the client requests to be processed. Sometimes requests are queued based on the client priority. Another group of coordination mechanisms orders the requests by their time of arrival.

Communication features describe the communication capabilities of the coordination mechanism. For the mechanisms analyzed, there are three child features to represent communication properties: storage capacity, direction of data flow and the volume of information transmitted (Figure 5).

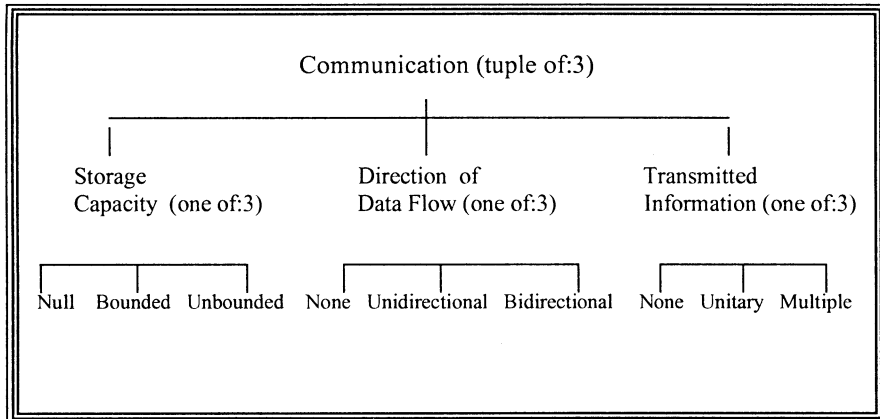


Figure 5. Communication Features

The storage capacity characterizes the capacity of the coordination mechanism for buffering the events or messages transferred between communicating partners. The capacity may be either null, bounded or unbounded.

The data flow direction indicates whether the data exchanged between client processes (those that use the same coordination mechanism) can be sent in one or two directions. When the mechanism is only a synchronization mechanism there is no data flow.

The size of information transmitted between two or more communicating partners can be none, unitary or multiple (more than one piece of information). When the size is variable, as it may be in the multiple case, it can be quite difficult to guarantee a predictable real-time behavior.

The operations provided by each of the mechanisms are described following a notation similar to the one used to describe the PPOOA components.

The implementation features denote those features related to the realization of the coordination mechanism for a particular environment.

The implementation issues of the coordination mechanisms that affect the design are those that have an impact on the dynamic behavior of the architecture. This means that they can affect the schedulability of the software architecture from a temporal point of view and may imply some missed deadlines. There are two issues to be considered:

- Avoidance of unbounded priority inversion. In the unbounded priority inversion, a low priority process uses a resource whereas a higher priority process is forced to wait for the resource for some period of time [13].
- Avoidance, when possible, of independent flows of execution (“threads”) in coordination mechanisms. Avoiding independent execution flows

(“threads”) when implementing coordination mechanisms decreases the overhead caused by context switching.

From a building point of view, the PPOOA vocabulary does not support aggregated coordination mechanisms because this would unnecessarily complicate the schedulability analysis.

5.1 Bounded-Buffer

The bounded buffer, also called buffer or queue of messages, is a temporary data holding area. Data producers and consumers make calls to send and get data from the bounded-buffer. Since the producers and consumers can run concurrently, the buffer has a predefined capacity for storing data.

The main features defined according to the PPOOA taxonomy are:

- Identification: Indirect.
- Synchronization: Exclusion (Internal State), Ordering (Temporal).
- Communication: Storage Capacity (Bounded), Direction of the Data Flow (One Direction), Transmitted Information (Multiple, Fixed Length).

5.2 General Semaphore

The general-semaphore is a non-negative integer value used as a synchronization mechanism. It is used to synchronize processes or control the access to a critical region.

The main features defined according to the PPOOA taxonomy are:

- Identification: Indirect.
- Synchronization: Exclusion (Internal State), Ordering (Temporal).
- Communication: Storage Capacity (Null), Direction of the Data Flow (Does not apply), Transmitted Information (None).

5.3 Mailbox

The mailbox is a mechanism that is used to pass messages, which may possibly have a variable length, between processes in an asynchronous mode.

The main features defined according to the PPOOA taxonomy are:

- Identification: Indirect.
- Synchronization: Exclusion (History), Ordering (Temporal and/or Priority).
- Communication: Storage Capacity (Null), Direction of the Data Flow (One Direction), Transmitted Information (Multiple, Variable Length).

5.4 Rendezvous

The rendezvous is a synchronous and unbuffered coordination mechanism that allows two processes to communicate bidirectionally. In this case, the processes may be implemented as Ada tasks.

The main features defined according to the PPOOA taxonomy are:

- Identification: Direct, Asymmetric.
- Synchronization: Exclusion (History), Ordering (Temporal).
- Communication: Storage Capacity (Null), Direction of the Data Flow (Bidirectional), Transmitted Information (Unitary, Fixed Length).

5.5 Transporter

A transporter can be considered to be an active data mover. The transporter makes calls to get and send messages from and to the coordinating partners (producer and consumer).

The main features defined according to the PPOOA taxonomy are:

- Identification: Indirect.
- Synchronization: Exclusion (History), Ordering (Temporal).
- Communication: Storage Capacity (Null), Direction of the Data Flow (One Direction), Transmitted Information (Multiple, Fixed Length).

6. UML METAMODEL EXTENSION TO INCLUDE PPOOA BUILDING ELEMENTS

UML stereotypes may be used to include the PPOOA vocabulary in the UML metamodel. The PPOOA profile has been created in order to extend the UML metamodel [14]. The PPOOA profile is built with the PPOOA elements that have been described above.

The UML Classifier is extended with other building elements that are characteristic of the PPOOA vocabulary. This is shown in the PPOOA Building Elements metamodel (Figure 6). A Classifier is a UML model element that describes behavioral and structural features. The Class element is a kind of Classifier that represents a concept within the system being modeled. It describes both the data structure and the behavior of a set of objects. A Class may provide or require interfaces.

In the PPOOA profile different kinds of classes are considered (Figure 6): *Active_Class*, *Domain_Component*, *Algorithmic_Component*, *Structure* and *Coordination_Mechanism*. Real-time domain independent attributes are depicted for each building element. These attributes are essential for applying time responsiveness assessment techniques, specifically RMA.

An `Active_Class` is a UML element. It is a class whose instances are active objects. Two types of `Active_Classes` are considered: `Thread` and `Controller`.

- `Threads` or light processes are treated in a special way, as they are divided in periodic and aperiodic. The `Periodic_Process` and the `Aperiodic_Process` are PPOOA metaclasses. As described previously, several real-time attributes such as priority, shared resources blocking time, offset and execution time per activity, have to be defined for each of the processes. The execution period in the periodic processes and the activation pattern in the aperiodic processes are also essential attributes that have to be determined during the architectural modeling of a real-time system.
- The `Controller` is a PPOOA element that is responsible for directly initiating and managing a group of activities that can be repetitive, alternative or parallel. Certain real-time attributes have to be defined: priority per thread, shared resources blocking time, execution time per activity and deadline.

Figure 6 shows how coordination mechanisms are a specialization of the UML “class” concept. New stereotypes have been created in PPOOA for these coordination mechanisms: `<<semaphore>>`, `<<mailbox>>`, `<<rendezvous>>`, `<<transporter>>`. These stereotypes will allow different iconic representations for these mechanisms.

used for real-time data acquisition, aerospace and telecom systems. An architecting process and design guidelines are also defined.

ACKNOWLEDGEMENTS

This work is partly supported by research funded from the European Union IST, 5th Framework Programme (IST-1999-20608).

REFERENCES

- [1] J.L. Fernández. *An Architectural Style for Object Oriented Real-Time Systems*. Fifth International Conference on Software Reuse. Victoria (Canada), IEEE 1998.
- [2] M. Frankel. *Analysis Architecture Models to ASG Models: Enabling the Transition*. Proceedings Tri Ada. 1992.
- [3] K. Jeffay, The Real-Time Producer/Consumer Paradigm: A Paradigm for the Construction of Efficient, Predictable Real-Time Systems. Proceedings of the *ACM/SIGAPP Symposium on Applied Computing*. Indianapolis, 1993.
- [4] A. Burns, A. Wellings. *HRT-HOOD: A Structure Design Method for Hard Real-Time Ada Systems*. Elsevier Science B.V. Amsterdam, 1995.
- [5] H. Gomaa. *Designing Concurrent, Distributed and Real-Time Applications with UML*. Addison Wesley/Pearson, Upper Saddle River, NJ, 2000.
- [6] B. Selic. *Turning Clockwise: Using UML in the Real-Time Domain*. Communications of the ACM Vol 42 No 10, October 1999, pp 46-54.
- [7] J.L. Fernández. *A Taxonomy of Coordination Mechanisms Used in Real-Time Software Based on Domain Analysis*. CMU/SEI-93-TR-34. Software Engineering Institute. Pittsburgh, 1993.
- [8] M. Shaw, D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice Hall. Upper Saddle River, New Jersey. 1996.
- [9] J. Warmer, A. Kleppe. *The Object Constraint Language. Precise Modeling with UML*. Addison Wesley Longman, Reading, MA, 1999.
- [10] G. Botella and J. Gosling. *The Real-Time Specification for Java*. IEEE Computer June 2000, pp 47-54.
- [11] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading MA, 1995.
- [12] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, A.S. Peterson. *Feature Oriented Domain Analysis (FODA). Technical Report CMU/SEI-90-TR-21*. Software Engineering Institute. Pittsburgh, November 1990.
- [13] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, M. González Harbour. *A Practitioner's Handbook for Real-time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, Dordrecht, 1993.
- [14] J.L. Fernandez, A. Monzon. *Extending UML for Real-Time Component Based Architectures*. International Conference on Software & Systems Engineering. Paris (France). December 2001.

Chapter 13

COTS Component-Based System Development *Processes and Problems*

Gerald Kotonya¹, Walter Onyino¹, John Hutchinson¹, Pete Sawyer¹ and Joan Canal²

¹Lancaster University, UK; ²CCS, Spain

Abstract: Component-based software engineering (CBSE) represents an emerging development paradigm based on assembling software systems from pre-fabricated components. This chapter discusses the promises and challenges of component-based system development. The authors propose a component-based system development process and use it as a basis for the discussion. Requirements engineering, system design, composition, verification and management are discussed in this context.

Key words: CBSE, processes, COTS software, requirements, design, composition, management

1. INTRODUCTION

Component-based software development (CBD) is being proposed as a means of reducing complexity and cost in software development. The drive to use components to construct software systems stems from a ‘parts’ philosophy derived from traditional engineering disciplines that promises instant productivity gains, accelerated time to market and lower development costs. However, software component technology is still immature and poses many problems for organisations intending to adopt it. Component-based system development proceeds by composing software systems from reusable components (often blackbox third-party software). However, the features supported by Commercial Off-The-Shelf (COTS) software components vary greatly in quality and complexity. Application contexts in which the components are used also vary considerably. This complexity together with

the variability in application domains means that specifications delivered with COTS software are likely to be incomplete or inadequate. In addition most COTS components are generally not “plug-and-play” and may require significant effort to adapt them to new situations.

The other more fundamental problem is that traditional software development models do not address the needs of component-based system development. Boehm [1] regards both the waterfall and evolutionary development model as unsuitable for component-based development for the following reasons:

- In the waterfall model requirements are identified at an earlier stage and the components chosen at a later stage. This increases the likelihood of the components not offering or supporting required features.
- Evolutionary development assumes that additional features can be added if required. However, the inaccessibility of component code prevents developers from adjusting them to their needs.

Many of these problems are unique to the componentware paradigm and are a consequence of the blackbox nature of COTS software and its volatility [2]:

- *Poor documentation.* COTS software components are almost always delivered as blackbox components with limited specification that focus on functional properties. This makes it difficult to predict how components might behave under different load conditions and contexts.
- *Poor support for integration.* Most component integration processes suffer from the lack of interoperability standards amongst different component models.
- *Limited adaptability.* COTS software components are generally not tailorable or “plug and play”. Significant effort is often required to build wrappers and “glue” between components in order to evolve applications and to tailor components to new situations.
- *Design assumptions.* The design assumptions of a COTS component are largely unknown to the application builder. In a situation where many complex functions are replaced by a single large-grain COTS component this may have serious implications for exception handling and critical quality attributes.
- *COTS volatility.* COTS software is subject to frequent upgrades. This often leads to a disparity in customer-producer evolution cycles and may result in unplanned upgrades being forced on the customer.
- *Vulnerability risk.* The use of COTS software introduces a vulnerability risk that may compromise system dependability (i.e. performance, reliability, availability, safety or security). This is particularly critical for safety-related and distributed systems where abnormal system behaviour or unauthorised system access may result in injury or loss of information.

- *Component mismatch.* COTS components in heterogeneous environments are likely to suffer from mismatches due to different data models, functional mismatches or resource conflicts.

These problems underpin the need for software engineering process models that can balance aspects of system requirements, business and project concerns, with the architectural assumptions and capabilities embodied in COTS software. The next sections describe the processes for developing component-based applications from COTS software, and discuss the problems associated with each development stage. Possible ways of addressing the problems are also suggested.

2. COTS-BASED DEVELOPMENT PROCESS

There are two broad development processes in component-based software engineering: component development and component-based system development (e.g. COTS-based software development). The component development process is concerned with the analysis of domains and development of generic and domain-specific components (i.e. *development for reuse*). To achieve successful software reuse, commonalities of related systems must be discovered and represented in a form that can be exploited in developing similar systems. Once reusable components are created, they can be made available on the open market as COTS software through distribution networks (i.e. component vendors and brokers). COTS software may be domain specific, product specific or generic. In general, the productivity increase from domain-oriented and product-oriented components is higher than from generic components (see Figure 1). However, the reusability of generic components is higher than that of domain-oriented and product-oriented components. Immature domains and those in which great variability exists have little scope for domain-oriented reuse, and application development may be limited to using generic components or custom development.

Component-based system development is the process of composing systems from pre-fabricated software components. The development process described here is mainly concerned with COTS software. Figure 2 shows our suggested 4-phase component-based application development process. It develops some of the early ideas on component-based development [1], [2] to provide a scalable process with a clear separation of concerns.

The negotiation phase attempts to find an acceptable trade-off amongst multiple (often) competing development attributes. The planning phase sets out a justification, objectives, strategies (methods and resources to achieve development objectives) and tactics (start and end dates, and tasks with duration) for the project.

The development phase implements the agenda set out in the planning phase. The first step in application development is the definition of system requirements. The requirements stage elicits, ranks and models the system requirements iteratively with the verification and negotiation and planning phases. The design stage partitions the requirements into sub-systems and eventually components, which can be replaced with COTS software. Like the requirements stage, the design stage proceeds in tandem with the verification, negotiation and planning phases, and may revisit the requirements stage from time to time. The composition stage replaces abstract design components with off-the-shelf “equivalents”. Off-the-shelf software components are packaged in many different forms (e.g. function libraries, frameworks and legacy applications). The composition process must devise mechanisms for integrating different components without compromising the system quality. Beyond this stage the system goes into a management cycle (discussed in Section 7).

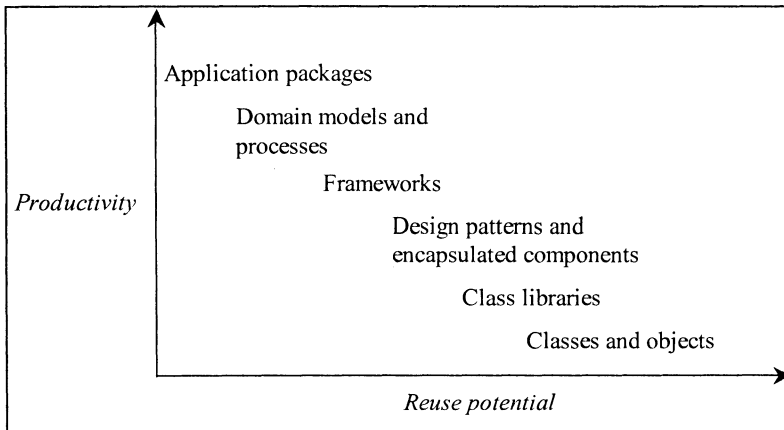


Figure 1. Productivity versus reuse potential

The verification phase is intended to ensure that there is an acceptable match between the COTS software and the system being built. This is important because perceptions of software quality may vary amongst component producers. In addition the blackbox nature of most COTS software diminishes the scope for correcting errors later in the system development. The verification phase varies in focus and detail across the development cycle. A matching colour scheme has been used to indicate the correspondence between different development stages and the aspects of verification that apply to them. At the requirements stage, verification is used to establish the availability of COTS software and viability of a COTS-based solution. At the design stage verification is concerned with ensuring that the design matches the system context (i.e. in terms of non-functional

requirements, architectural concerns and business concerns). This may require detailed blackbox testing of the COTS software and architectural analysis. At the composition stage verification translates to design validation, through component assembly and system testing.

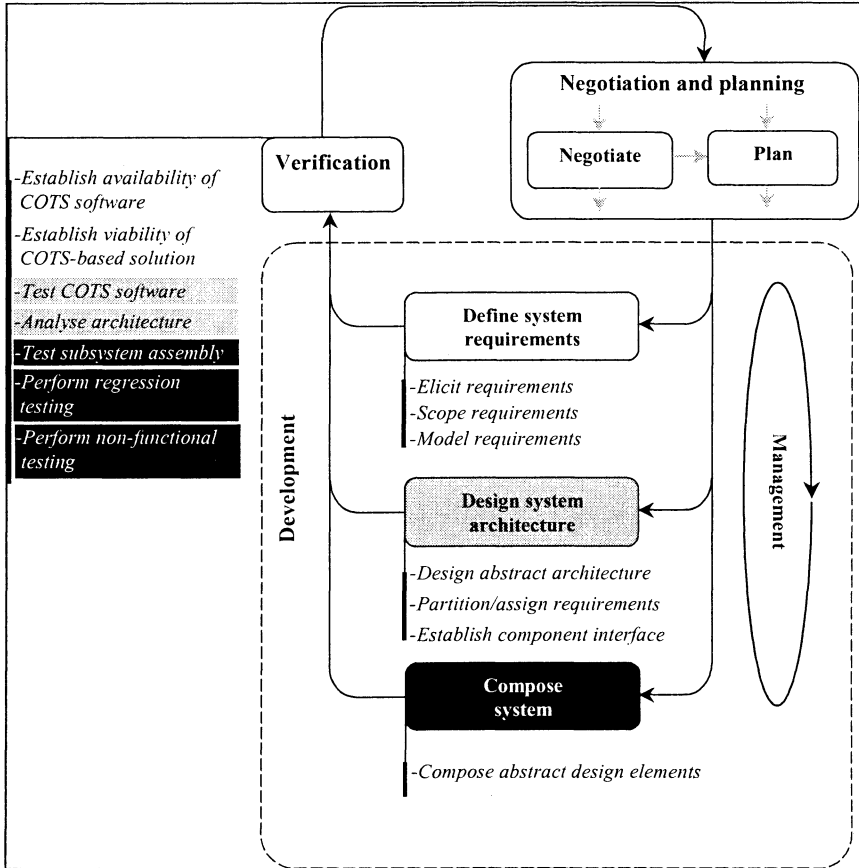


Figure 2. COTS-based development process

3. REQUIREMENTS ENGINEERING

It is generally acknowledged that good requirements engineering is essential for successful component-based system development [3]. However, few methods address the problem of how requirements for component-based systems are derived. Current approaches are based on procurement models in which the requirements process is driven by the availability of COTS software. In one such a model, Vidger [4] proposes that system requirements

should be defined according to what is available in the marketplace, and that organisations should be flexible enough to accept COTS solutions when they are proposed. Vidger notes that overly specific requirements preclude the use of COTS components and should be avoided. This is a reasonable approach, however, most systems have requirements that are unavoidably ‘specific’, for example, critical systems. Critical systems are likely to have stringent, often competing quality requirements (e.g. performance, efficiency, safety, reliability etc.).

The PORE (Procurement-oriented requirements engineering) method [3], proposes an approach in which the requirements process is tightly integrated with a process for selecting COTS products. Central to the PORE approach is an iterative process in which candidate products are tested for fitness against increasing levels of requirement detail. Products that fail to meet requirements are rejected at various levels. While the PORE approach provides insight into product evaluation processes, its singular focus on COTS component selection has been criticised for ignoring system level concerns and the important role architecture plays in formulating the requirements for component-based systems [5]. The approach used in PORE makes it difficult to address system-level concerns and reduces the scope for requirements negotiation. Secondly, components are often intended to “plug” into existing or pre-defined architectures. It is important that architectural considerations are taken into account as they constrain the way a new component connects and interacts with other system components, and form the basis for assessing the impact on the system of new components.

Proper requirements formulation is critical to the success of component-based systems. However, requirements methods must recognise that in addition to conventional stakeholder requirements problems, component-based systems also present problems that are unique to the componentware paradigm. The principal challenge in requirements engineering for component-based systems is to develop requirements models that allow us make the best use of the available COTS technology by balancing aspects of requirements with the architectural assumptions embodied in COTS software and the capabilities of COTS software.

A competent requirements engineering process for component-based systems must address the following issues:

1. Provide a means for eliciting requirements. Component-based systems development is primarily a problem of integrating black-box components rather than creating components. It is essential that requirements for the environment in which the intended system is going to work be clearly understood. This often involves understanding the requirements of the target system from actor, stakeholder and system perspectives. Actors correspond to system operators and existing software components that interact directly with the target system. Stakeholders include specific

departments of the user organisation (e.g. finance, training etc), existing legislation, COTS certification organisations and COTS vendor. A System perspective is the embodiment of global system requirements that the proposed system must conform to. These include dependability (i.e. availability, security, performance, reliability and safety) and system resource requirements.

2. Provide a means for ranking and scoping requirements. Requirements ranking provides for incremental development and early validation of COTS products. The requirements method should provide a mechanism for identifying, prioritising and scoping requirements at different levels of abstraction.
3. Provide “hooks” to a process for identifying and evaluating off-the-shelf software components. Early evaluation is useful for establishing the availability and capabilities of the COTS software.
4. Provide a mechanism for mapping requirement specifications to COTS capabilities. It is also important that the scheme provides a mechanism for supporting negotiation. This is because in practice the desired level of COTS software capability and quality is rarely achieved. For certain critical requirements, a COTS solution may not be adequate or even appropriate.

Finally, it is important to mention that requirements definition and system design are not linear but highly interlaced activities. There are several reasons for this:

- The system being specified is likely to be part of an environment made up of other COTS software components. The components in the environment are likely to impose requirements on the system being specified and to constrain the system design.
- For non-trivial systems, some architectural design is often necessary to identify sub-systems and their relationships. The requirements for the sub-systems may then be specified.
- Reasons of budget, schedule or quality may force an organisation to reuse particular software components in constructing the new system. This constrains both the system requirements and the design.
- System requirements are subject to change when the development confronts the reality of design (e.g. need to trade-off requirements to resolve competing quality attributes) and limited budgets.

4. SYSTEM DESIGN

Component-based system design describes how the components that make up the system interact to deliver the required functionality, and how the design process reasons about the appropriateness of particular components

and services in different contexts. Different COTS products may result in different system architectures to achieve desired quality attributes because of the different constraints they impose on the system design [6]. A design trade-off may, for example, require that critical components interact only through “validation” components to ensure that the desired level of system security or safety is maintained, even if this means a loss in system performance.

The design process starts with the partitioning of the system requirements (services and constraints) into logical “components” or “sub-systems”. Figure 3 shows how a top-down process may be used to partition the system by clustering services to reflect desired functionality and quality attributes. The initial partitioning is driven by architectural considerations (e.g. safety, performance, security, availability etc.) and may be supported by design patterns that lend themselves to those quality considerations.

Subsequent partitioning is subject to a negotiation process that takes into account business concerns, architectural considerations and COTS software considerations. The design process is supported by verification to ensure an acceptable match between the COTS software and the desired system features. Verification may involve any or all of the following activities:

- Tests performed to verify COTS software functionality.
- Architectural analysis to establish how well the design supports desired quality attributes (e.g. performance, security and availability etc).
- In certain cases where matching COTS software cannot be found (e.g. incompatible interface, incompatible data model etc.), a decision may have to be made to accept the component and to correct the anomaly using “glue code”).
- In extreme cases of incompatibility, it may make better design sense view parts of the architecture as a “place-holders” for custom developed components.

In order to support the design process we need mechanisms that allow the designer to define architectural elements and their relationships, and to support their evolution through levels of abstraction. The Catalysis method [7] is one of a few of software design methods that explicitly supports component-based system design. The Catalysis method is based on the Unified Modelling Language (UML) and focuses on designing the basic interface signature that characterises the component functionality. A basic component interface signature comprises *provided* and *required* services interfaces, and an *events* interface. A component service interface defines its functional capability. An event interface describes the sequence of events associated with the provision of a component service.

Formal Architecture Description Languages (ADLs) are also emerging as an effective way of describing component-based system designs. ADLs provide a formal mechanism for representing software system architectures, and as such they address the shortcomings of informal representations. In

addition, sophisticated ADLs allow for early analysis and feasibility testing of architectural design decisions [8], [9]. However, ADLs vary widely in their ability to support the specification of particular architectural styles, variability of the same architecture, support for mapping abstract system designs to off-the-shelf components and support for analysis.

An effective component-based design language should address the following issues:

1. Provide a mechanism for partitioning the system requirements into a set of independent sub-systems and components. For complex systems, this activity may start at the requirements stage and develop during system design. Architectural partitioning and requirements allocation is a critical activity as it provides a means of structuring and organising the specification. It forms the starting point for a detailed specification of the system.
2. Component-based design proceeds by evaluating candidate off-the-shelf components and qualifying them according to the desired system requirements, critical quality attributes, adaptability and architecture. There are two important aspects component-based system design that should be addressed by a good design method:
 - An effective design process should allow the engineer to decide which COTS products are feasible (cost, technology, support, availability etc) and which COTS products fit the design goal.
 - The main objective of designer is to achieve “fitness for use”. Fitness is a property that is achieved when the COTS product has an acceptable match with the context in which it is going to operate. For many systems there is often a need repair a design “misfit” through component adaptation or custom development to resolve problems of resource contention, quality requirements or architectural assumptions. There is a need for design techniques that can allow the designer to evaluate the extent of “fitness” of a component or group of components in a context.
3. Provide a mechanism for distinguishing between component interfaces. A component interface signature characterises the component functionality. A basic component interface signature comprises the *provided* and *required* service interfaces. In addition to its interface signature, the properties and operations of a component interface may be subject to a number of semantic constraints regarding their use. In general, there are two types of such constraints: those on individual elements and those on concerned with the relationships among the elements. Examples of the first type are the definition of the operation semantics (e.g. in terms of pre-/post-conditions) and the range constraints on properties. An example of the second type is inter-relating properties in terms of their value settings.

In designing component interfaces it is important to distinguish between direct and indirect interaction:

- *Direct interaction.* A specification of the interface protocols that describes direct component interaction. This refers to cases where a component knows of the existence of other components and directly invokes one or more of their services.
 - *Indirect interaction.* A specification of interface protocols that describe indirect component interactions. This refers to interactions that take place through a standardised middleware (e.g. DCOM or CORBA) or kernel. This refers to cases where components can inquire about the possible supported services and use them without knowing where the other component is located.
4. Component-based design proceeds by evaluating candidate off-the-shelf components and qualifying them according to the desired system requirements, critical quality attributes, adaptability and architecture. There are two important aspects component-based system design that should be addressed by a good design method:
- An effective design process should allow the engineer to decide which COTS products are feasible (cost, technology, support, availability etc) and which COTS products fit the design goal.
 - The main objective of designer is to achieve “fitness for use”. Fitness is a property that is achieved when the COTS product has an acceptable match with the context in which it is going to operate. For many systems there is often a need repair a design "misfit" through component adaptation or custom development to resolve problems of resource contention, quality requirements or architectural assumptions. There is a need for techniques that can allow the designer to evaluate the extent of “fitness” of a component or group of components in a context.

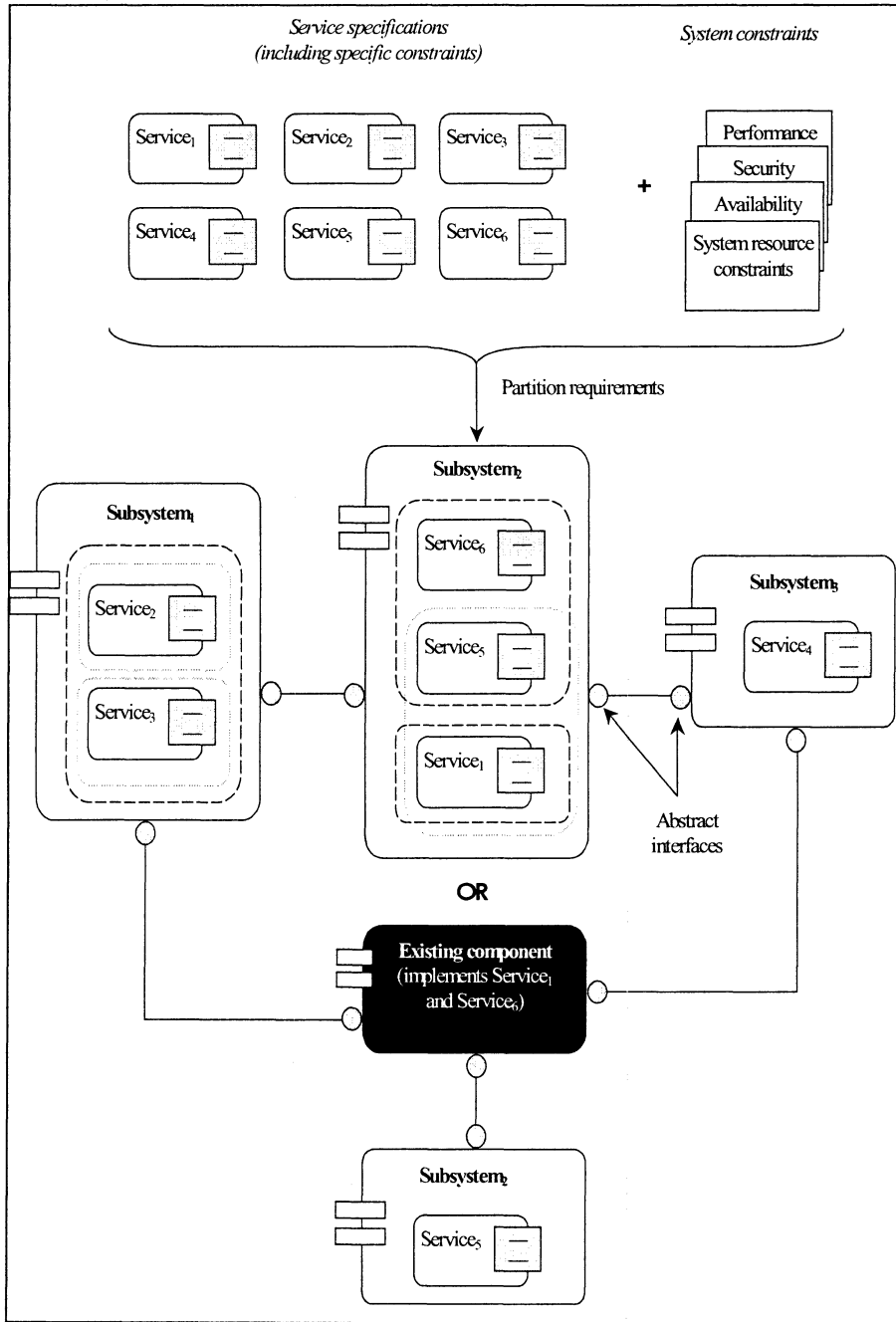


Figure 3. System design

5. SYSTEM COMPOSITION

System composition proceeds by replacing abstract design level components with COTS software. As in design, the composition process may be supported by an Architecture Description Language [9] that specifies business, architectural and component-level constraints that COTS software must satisfy before a replacement can occur. At this stage the use of ADLs also provides useful traceability back to requirements.

Individual COTS components are developed to meet different requirements based on different assumptions about their application contexts. It is therefore common practice to adapt or tailor components for use in new systems and environments. The extent to which components can be adapted depends on the degree to which the component is configurable and the extent to which its internal structure is accessible to the programmer. For “difficult” cases, the integration process may make use of “gluing technology”, which may be unrelated to the components, to provide an interface between components. Scripting languages such as VBScript and JavaScript are commonly used as gluing technology to manipulate collections of components, which expose interfaces that conform to particular scripting architectures (i.e. scriptable components). The idea of scripting components extends to the construction of applications from several off-the-shelf-components to the manipulation of applications made up of several components (e.g. graphics, spreadsheets etc.).

Understanding how components are packaged and delivered is central to successful composition. Components are packaged and delivered in many different forms [4]. These forms are closely related to the reuse/productivity potentials shown in Figure 1. Forms of component packaging include:

- *Function libraries.* This is probably the most common way of packaging components. Components are delivered as a set of library routines, which are linked at build time.
- *Legacy applications.* An application is part of an organisation's structure and workflow and is included as a component with the new system.
- *Off-the shelf applications.* A component may be delivered as a stand-alone application (which may or may not have open interfaces and data formats). Integration can take different forms, such as API calls, shared data in standard formats, event passing, drag-and-drop, etc. The most common form of integration for these kinds of component is shared databases and shared files.
- *Tools.* Examples include graphical user interface builders and component development environments. The tools typically work by having the developer describe the system using the tool's notation. The tool generates source code that can be compiled and linked with other components of the system.

- *System services.* Operating systems, databases, windowing systems, and device drivers are typically purchased as COTS components.
- *Application frameworks.* Components may be packaged as replaceable parts of a component framework. A framework is designed to be extensible and to be integrated with other frameworks.

Appropriate component integration mechanisms are essential for successful application development. The integration mechanism used depends largely on how the component is packaged or delivered. There are four main mechanisms for integrating COTS components:

- *Procedure calls.* The COTS component is accessed by linking to a procedural interface. Examples include components that are packaged as function libraries, applications with an API, and databases with an SQL interface.
- *Desktop supported capabilities.* Desktops provide limited capabilities for integrating components through features such as drag-and-drop, clipboards, cut-and-paste etc. Office automation software is generally integrated this way.
- *Data sharing.* For applications that store data in a standard format, integration is normally accomplished by having components read and write each other's data. The shared data is usually stored in files or in a shared database.
- *Frameworks.* Frameworks can be tailored and extended in several ways:
 - i. *Plug-ins.* Developers can add functionality to application by writing or purchasing a commercial off-the-shelf "plug-in". A plug-in notifies the framework of its capabilities and services and the framework calls the plug-in as required.
 - ii. *Scripting.* A script is an executable fragment of code, which is dynamically linked to components of the system. A script can be used to extend the behaviour of a component (by having the component execute the script), or it can be used as a coordination mechanism to integrate two or more components (by providing the "glue" for linking the components together).
 - iii. *Inheritance.* Inheritance allows specific parts with a component to be specialised and modified.

The composition process may be affected by several factors including; poor procurement and evaluation schemes, lack of adequate documentation, different vendor-customer evolution cycles and lack of interoperability standards and competing architectural requirements. Many of these issues are also related to the long-term management of the system.

6. COTS VERIFICATION

The risk of composing systems from COTS components of unknown reliability and the severely limited ability of the system integrator to modify COTS components make component and system testing a critical aspect of component-based development. Voas [10] suggests that the risk posed by unreliable components should lead developers to think in terms of disposable software systems. He observes that as we move toward component-based software systems, quality will play a key role in the maintainability of component-based systems. Independent certification of COTS components has also been suggested as a way of addressing the problem, particularly for critical systems [11]. We regard component-based system testing as a process that includes system verification and validation. The process starts from early system formulation through to system composition. It includes the evaluation of COTS software, the assessment of the impact of system changes and design trade-offs, and the testing of the integrated system.

COTS software testing is motivated by several factors:

- *Black-box nature.* The black-box nature of COTS components means that the system integrator has no access to the design considerations or source code.
- *Perception of quality.* The perception of software quality may vary across COTS software vendors and application domains. For many vendors the time-to-market may be more important than delivering high-level configurability, reliability, performance and other software qualities.
- *Extraneous features.* A new version of COTS software is likely to contain new features that are not used by the system. There is a risk that unused features may still have some indirect impact on the system behaviour. This risk can complicate testing if the COTS software must be tested for even those functions or features that are not directly used by the system. The explosion of the Ariane 5 rocket [12] is a classic example in which design assumptions and untested software features can result in a costly disaster. Ariane 5 rocket failed because it included software ported from an early version of the rocket (Ariane 4) that performed a computation that was necessary early versions of the rocket but not in Ariane 5. The computation caused an exception during the Ariane 5 flight, which was not caught.

There are four different scenarios that can be encountered when testing component-based systems [13]. These scenarios involve testing individual components as well as the systems composed from the components:

- *Prior to deployment.* The system integrator needs to thoroughly test a new COTS component prior to deploying it in a larger system.

- *Integrated system.* If a new component is added to the system or an older version replaced, the integrated system must be tested. Testing should also be done if the system configuration is altered.
- *Regression testing.* It is a good idea to perform regression testing on selective critical system components whenever new versions of other constituent components are installed in the system.
- *Non-functional testing.* Various kinds of non-functional testing on the system are required to ensure that the system meets the desired level of performance, dependability, stress and loading.

The testing of COTS components is constrained by the lack of source code. Therefore, the system integrator restricted to performing only blackbox testing. This process is further complicated by the fact that most COTS components are supplied with limited documentation.

Harrold [12] suggests that the COTS software vendor should make a summary of the test and analysis information available with COTS software, to facilitate further analysis and testing by the system builder. The information should be represented in a standard notation independent of the language in which the component is implemented. The component should provide suitable query facilities (for example, methods or operations) to retrieve the summary information.

The process of testing off-the-shelf components is complicated by several factors:

- i. *Poor specification.* The blackbox nature of COTS components means that the system integrator can only perform blackbox testing. However, the lack of detailed component specification diminishes the quality of testing that can be done.
- ii. *Enterprise heterogeneity.* Different, possibly competing vendors may supply COTS components. Complex licensing arrangements would mean that no one vendor has complete control over the development artifacts associated with each component for the purposes of testing.
- iii. *Technological heterogeneity.* In a distributed environment, different components may be developed for different hardware and operating system platforms. This would require that the testing method work on multiple hardware and operating system platforms (possibly multiple programming languages).
- iv. *Repair response time.* Often vendors and/or developers of the COTS software have to be involved both in debugging the software and in making repairs. This means that the response time for repairs is likely to be determined by the vendor(s), not the system management team.
- v. *Test adequacy assessment.* A major problem with testing component-based systems is the lack of a sufficient theoretical basis for assessing the adequacy of the tests.

To address these problems, component-testing regimes should serve six aims:

1. *Discovery.* The process should expose undocumented features and/or faults in the COTS software.
2. *Verification.* The test process should verify producer data/specification.
3. *Fitness for purpose.* The process should establish how well the component capabilities fit in with the system needs.
4. *Masking.* The process should establish the extent to which it is possible to mask out unwanted component features.
5. *Adequacy.* The process should set realistic test adequacy criteria that take into account the resources available and criticality of the component being tested.
6. *Early validation.* The test process must aim at early component verification and validation to minimise repair delays caused by slow repair-response times.

7. MANAGEMENT CHALLENGES FOR COMPONENT-BASED APPLICATIONS

The maintenance and extended development of a component-based application poses many risks to the customer. The nature of the risk varies with the nature of the development process, application domain, system design characteristics and the choice of COTS software used [4]. These risks are related to:

- *Different producer-customer evolution cycles.* Commercial software component vendors and application developers are driven by different goals and objectives. The disparity in the customer-vendor evolution cycles may impose on the customer unplanned component upgrade requirements that may impact adversely on the organisation and system.
- *Funding risk.* The uncertainty about how often COTS components in a system may have to be upgraded or replaced, and how much more of the system may have to be changed as result, makes it difficult to plan and predict costs over the life cycle of a system. In mixed vendor environments, variations in licensing agreements may, in addition, lead to wide variances in the cost of using COTS software.
- *Vulnerability risk.* Because of their blackbox nature, the use of COTS software introduces a vulnerability that may compromise system dependability.
- *Upgrade risk.* Upgrading to a new version of COTS software poses several risks:
 - i. Hidden incompatibilities may cause unforeseen side effects in the system necessitating a complete system update.

- ii. A new version of COTS software may have data formats that require changes to be made to the formats and contents of existing files and databases created by prior versions of the COTS software.
 - iii. Changes in the quality attributes of a new version of COTS software (e.g. performance, security, safety, reliability etc.) may be incompatible with the user requirements. This may adversely affect the operational capabilities of the system.
 - iv. A new version of COTS software may provide additional capabilities that may have to be suppressed or restricted due to security concerns.
 - v. The new version of COTS software may be incompatible with the existing hardware or operating platform.
 - vi. The introduction of new hardware may force changes to interfacing components, which may ripple through the system.
 - vii. Changes in the memory, processor and operating requirements for a new version of COTS software may be incompatible with the existing hardware and operating system.
- *Maintaining quality.* Maintaining system quality involves ensuring that any change to the system does not compromise its quality unacceptably. It also means identifying sources of errors, repairing them and assuring that the system is error-free. However, the blackbox nature of COTS components and vendor priorities combine to make this a difficult task.
 - *Configuration management.* Configuration management poses two main risks for component-based systems.
 - i. New versions of components may have to be installed frequently.
 - ii. It may be easy for maintainers at different sites of one system to obtain replacement or upgrades for COTS software directly from vendors without following configuration management procedures.

System management processes should aim at addressing the following problems:

1. *Asset management.* Component-based system management processes must provide a framework for managing the acquisition, usage and evolution COTS products. The inventory of COTS products, versions, where they reside, and the financial obligations licenses associated with them, are a critical aspect of system management.
2. *Upgrade impact analysis.* It is impossible to determine the cost and difficulty of upgrading COTS software without proper analysis. The system management process must consider the different ways that a COTS product might cause changes to the operational system, the software in the system, and the software maintenance process.
3. *Quality control.* Maintaining quality in component-based systems can be a complex and expensive activity. There is need for cost-effective quality

control methods that address the problem of the fault identification, repair, and the tracking of system fixes.

4. *Configuration management.* Configuration management is an essential part of system management and component-based systems are no different in that respect. The management process should provide a framework for tracking and controlling the versions of COTS products and custom software installed at all locations for the system.
5. *Market research.* The different vendor-customer evolution cycles present a particularly difficult system management problem. Vendor evolution cycles are largely driven by market changes. It is therefore important that the management process provides a framework for factoring market research into the process.

8. CONCLUSION

This chapter has provided an overview of the CBSE process and the problems posed by the development paradigm at different stages of system development. Component-based development is a highly iterative process requiring simultaneous consideration of the system context (system characteristics such as requirements, cost, schedule, operating and support environments), capabilities of the COTS products in the marketplace, viable architectures and designs. The nature of COTS software and how it relates to these aspects has been discussed. We have identified the challenges and problems likely to be faced by component-based system developers and outlined various ways to address the problems. The importance of verification has been emphasised and a detailed discussion of the management challenges of component-based systems provided. We accept that some of the proposed solutions are likely to require further research before their effectiveness can be ascertained. Equally, we believe that these problems must be addressed if component-based system development is to become a successful software development paradigm.

REFERENCES

- [1] Boehm, B. and Abts, C. (1999) COTS Integration: Plug and Pray, *IEEE Computer* 32(1): 135-138, Jan. 1999.
- [2] Brown, A.W and Wallnau, K.C. (1998) The current state of CBSE, *IEEE Software*, 15(5), 1998
- [3] Ncube, C. and Maiden, N (1999) PORE: Procurement-oriented requirements engineering method for the component-based systems engineering development paradigm, *Proc. 2nd IEEE International Workshop on Component-Based Software Engineering*, Los Angeles, California, USA, May, pages 1-12, 1999.

- [4] Vigder, M., Gentleman, M. and Dean, J.(1996) COTS Software Integration: State of the Art, Institute for Information Technology, National Research Council, Canada, 1996.
- [5] Kotonya, G., Hutchinson, J., Onyino, W. and Sawyer, P (2002) Component-Oriented Requirements Expression, To appear in Proc. of 16th European Meeting on Cybernetics and Systems Research, Vienna, Austria, April 2002.
- [6] Heineman, G.T. (1998) A model for designing adaptable software components, Proc 22nd Annual International Computer Software and Applications Conference, pg.121-127, Vienna, Austria, 1998
- [7] D'Souza, D. F. and Wills, A. C., (1998) Objects, Components, and Frameworks With Uml: TheCatalysis Approach, Addison-Wesley, 1998.
- [8] Shaw, M. and Garlan, David.(1996) Software Architectures Perspectives on an Emerging Discipline, Prentice-Hall, 1996.
- [9] Medvidovic, N. and Taylor, R.N. (2000) A Classification and comparison Framework for Software Architecture Description Languages, Trans. IEEE Software Eng., 26 (1), January 2000, pp.70-93.
- [10] Voas, J.M. (1998) The challenges of using COTS software in component-based development, Computer, 31(6), 1998, p.44
- [11] Dean, J., (1999) Timing the Testing of COTS Software Products, *International Workshop on Testing Distributed Component-based Systems, ICSE'99*, Los Angeles, California, May 1999.
- [12] Harrold, M.J., Liang, D. and Sinha, S. (1999) An Approach to Analysing and Testing Component-based Systems, *International Workshop on Testing Distributed Component-based Systems, ICSE'99*, Los Angeles, California, May 1999.
- [13] Rosenblum, D.S., (1997) Adequate Testing of Component-based Software, *Technical Report No. 97-34*, University of California, Irvine, August 1997.

Chapter 14

Component-Based Software Measurement

Yingxu Wang

*Theoretical and Empirical Software Engineering Research Center, University of Calgary,
Canada*

Abstract: Software components and component-based systems can be measured by architectural and functional sizes and complexities. The current approaches to functional measurement, such as function points, do not consider the internal structures of components; while other approaches to architectural measurements, such as the McCabe cyclomatic metric, do not consider I/Os of a system. This chapter introduces a new measurement approach, the equivalent functional size (EFS), to the measurement of both architectural and functional attributes of component-based software systems. The physical meaning of an equivalent functional unit (EFU) of software is explored for the first time, and the basic control structures (BCSs) of software are adopted to model the internal structure and complexity of a component-based software system. This work demonstrates that a software measurement system can be described in a formal and algebraic way. Thus, new measures may be derived based on existing and well-defined measures.

Key words: Software engineering, component-based software, measurement, metrics, quantitative models, SEMS, formal description

1. INTRODUCTION

Software engineering metrics and measurement are important technologies towards quantitative software engineering. It is recognized that “the history of science has been, in good part, the story of *quantification* of initially *qualitative* concepts.” [6]. In this chapter *software metrics* are defined as a standard or commonly accepted scale system, which defines the measurement of software attributes and their units and scopes. *Software measurement* is defined as a comparing process that quantitatively assesses a common

attribute of software entities against a given scale of metrics. In addition, *measure* of software is defined as a relation between an attribute and a measurement scale.

A variety of measures and metrics were developed towards the measurement of object-oriented software [1, 7-9, 11, 14]. Some well-defined proposals are such as Chidamber and Kemerer's metrics (CKM) suite [8], the Goal/Question/Metric (GQM) paradigm [3, 4], Henderson-Sellers' OO complexity metrics [11], Zuse's software measurement framework [24], and the Software Engineering Measurement System (SEMS) [18-21]. CKM is a set of six metrics for object-oriented software measurement, including weighted methods per class (WMC), number of children of a class (NOC), depth of inheritance tree (DIT), coupling between objects (CBO), response for a class (RFC), and lack of cohesion in methods (LCOM). However, CKM is focused only on the architectural attributes of OO software, and it is not a complete model for measuring the OO architectures [8, 11, 19].

GQM is a goal-oriented measurement paradigm for software engineering developed from the 1970s to 1980s [3, 4]. The GQM method provides an inference mechanism for deriving measures from measurement goals. A GQM measurement framework can be developed by setting a number of business goals, asking questions on achieving these goals, and deciding metrics for measuring the achievement of these goals. However, GQM is a typical empirical method, neither the goal, question or metric is formally defined or ensured to be quantitatively measurable. Since metrics that may be derived are dependent on the number and quality of questions, there is no defined way to determine if the questions on a given goal are complete, correct and verifiable. Furthermore, since all metrics may be described by natural language, inconsistency and ambiguity of metrics are common problems in implementing a GQM framework.

It is recognized that software engineering measurement is a systematic activity rather than separated individual activities [17-21]. A software engineering measurement system (SEMS) has been developed [18-20] to address the practical requirement in quantitative software engineering. SEMS consists of more than 300 formally defined software product measurement, software engineering process measurement, and software engineering project predictive and estimative measurement. As a comprehensive software engineering measurement system, SEMS is designed to support different approaches to software engineering measurement such as goal-oriented [3, 4, 21], process-oriented [17], and project-oriented measurement [17, 21].

This chapter explores architectural characteristics of component-based software (CBS) and their measurability. A set of common attributes of CBS, which is measurable and comparable independently from software projects and programming languages, is formally modelled and described [18, 19]. The measurement framework for CBS covers component and system

measures of physical sizes, equivalent functional sizes, complexity and common architectural attributes.

2. MODELS OF COMPONENT-BASED SOFTWARE SYSTEMS

Definition 1. A software system, S , can be described as a graph:

$$S = \langle N, R \rangle \tag{1}$$

where N is a set of elements modelled by nodes of the graph, and R a set of relations modelled by edges.

Definition 2. A component, C , of a software system is modelled by a subgraph that represents a set of nodes and their relations within the software system S , i.e.:

$$C = \langle N_C, \{R_C \cup R'_C\} \rangle \tag{2}$$

where $C \subseteq S$, $N_C \subseteq N$. R_C denotes the internal relations of the component, $R_C \subseteq N_C \times N_C \subseteq R$, and R'_C represents the external relations of the component, $R'_C \subseteq R$, and $R'_C = \{ \langle e_1, e_2 \rangle \mid (e_1 \in N_C \wedge e_2 \notin N'_C) \vee (e_1 \notin N_C \wedge e_2 \in N'_C) \}$. A component-based system is a special software system in which all the nodes are exclusively partitioned into different components. That is, the elements of components do not overlap in a component-based system, but there may be external relations across components. An example CBS is shown in Figure 1, where the CBS consists of 3 components C_1 to C_3 , and each component has its own internal (R_C) and external (R'_C) relations.

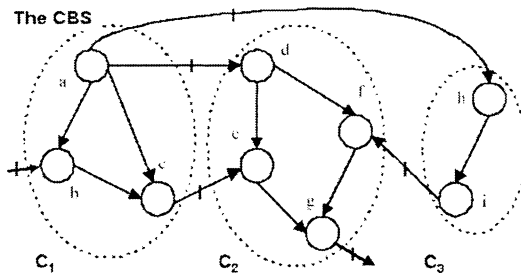


Figure 1. A sample component-based system (CBS)

Definition 3. A component-based system, \hat{S} , can be defined by a graph that consists of a set of components, C , and a set of external relationships, R' , between the components, i.e.:

$$\hat{S} = \langle C, R' \rangle (3)$$

where, for a given component C_i , $N_C \cap N'_C = \phi$, if $N_C \notin C_i \wedge N'_C \in C_i \wedge C_i \in C$; and $R'_{C_i} \subseteq R'$, if $C_i \in C$.

3. MEASUREMENT OF PHYSICAL SIZES OF SOFTWARE COMPONENTS

Software sizes can be classified into three basic categories: the *physical size*, *memory size*, and *functional size*. This section describes the measurement on physical sizes of software components. Memory sizes of components are simply the information bytes of their source or object code. The measurement of functional sizes will be developed in Section 4.

The physical size of a CBS can be formally defined at the method, class, component, and system levels.

Definition 4. The *physical size of a method*, S_{p-m} , is a meta measure that can be directly counted by:

$$S_{p-m} = \# (\text{source lines of code}) [\text{LOC}] \quad (4)$$

where # is the cardinal calculus that counts the number of lines of source code in a method. It is noteworthy that the unit of S_{p-m} is LOC, while the name of the measure is “the physical size.”

Definition 5. The *physical size of a class*, S_{p-cl} , is defined as the sum of those of the n_m methods included in the class. i.e.: $S_{p-cl} = \sum_{i=1}^{n_m} s_{p-m}(i)$ [LOC] (5)

Definition 6. The *physical size of a component*, $S_p(C)$, can be defined as the sum of the physical sizes of all n_{cl} classes comprising the component, i.e.:

$$S_p(C) = \sum_{j=1}^{n_{cl}} S_{p-cl}(j) = \sum_{j=1}^{n_{cl}} \sum_{i=1}^{n_m} s_{p-m}(j, i) [\text{LOC}] \quad (6)$$

Definition 7. The *physical size* of a component-based system \hat{S} , \hat{S}_p , is the sum of all sizes of its n components, i.e.:

$$\hat{S}_p = \sum_{k=1}^n S_p(C_k) \quad (7)$$

The property of \hat{S}_p is non-negative, i.e.: $\hat{S} \geq 0$.

This section dealt with the physical sizes of software components and component-based systems, from the bottom up, at the method, class, component, and system levels. A supplement measure on functional sizes of software will be developed in next section.

4. MEASUREMENT OF FUNCTIONAL SIZES OF SOFTWARE COMPONENTS

The fundamental problems in measuring functional sizes of software have long been investigated and yet to be solved. Because functional sizes have the advantage of language and implementation independence, they are more important and useful than the physical sizes of software, particularly for COTS and in-house developed components.

A conventional measure of functional sizes is the function point [2]. The function point method was developed in the context of information systems and it only considered system inputs and outputs (I/Os), and adopted a complicated weighting system for a dozen factors. A major conceptual gap in function points is that the physical meaning of what is one function point is not clear and intangible.

A new software functional measure, the *equivalent functional size* (EFS), is introduced in this section, in order to describe the fundamental attribute of software functionality [19, 20]. Using the EFS, the size of a software component or system can be determined quantitatively. Therefore, the following tricky questions in measuring component-based software can be solved:

- a) What is the functional size of a component?
- b) Is there a basic measurement unit for software components in CBS?
- c) Is a given component a primary (unity) component or a complex component?
- d) How to measure and compare the functionality and complexity of two similar commercial off-the shelf (COTS) components from different vendors?

4.1 Basic control structures (BCSs) of component-based software

In contrast to function points, the concept of EFS is not only based on I/Os of a component, but also takes into account of its internal structures known as the basic control structures.

Definition 8. The *basic control structures (BCSs)* are a set of essential flow control mechanisms that are used for building logical structures of software.

There are three BCSs commonly identified: the *sequential*, *branch*, and *iteration* structures [23]. Although it can be proven that an iteration can be represented by the combination of sequential and branch structures, it is convenient to keep iteration as an independent BCS. In addition, two advanced BCSs in system modelling known as *recursion* and *parallel*, have been modelled by Hoare [12]. Wang [22] extended the above set of BCSs to cover *function call* and *interrupt*.

The Seven categories of BCSs described above are profound architectural attributes of software systems. These BCSs and their variations are modelled and illustrated in Table 1, where their equivalent functional weights (W_i) for determining a component's functionality and complexity are defined. Formal definitions of BCSs in the real-time process algebra (RTPA) have also been provided in Table 1 [22].

4.2 The equivalent functional size (EFS) of software components









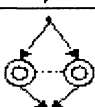
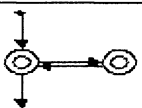
A component's EFS is considered proportional to its numbers of inputs and outputs, as well as to the complexity of its internal BCSs [18, 20]. In the other words, EFS is a function of these three factors, i.e.: $EFS = f(N_i, N_o, W_{bcs})$. Thus, an equivalent functional unit of software can be defined as follows:

Definition 9. The *equivalent functional unit (EFU)* of a software component, s_{fo} , is defined as the simplest component with one input ($N_i=1$), one output ($N_o=1$), and one unit of structural weight ($W_{bcs}=1$), i.e.:

$$S_{fo} = f(N_i, N_o, W_{bcs}) = N_i \times N_o \times W_{bcs} = 1 \times 1 \times 1 = 1 \text{ [EFU]} \quad (8)$$

Formula 8 models a tangible and primary unit of software functional size. It is intuitive that the larger each of the above factors, the larger the EFS. Based on this, EFS can be derived as a function of the three factors as shown below.

Table 1. Definitions of BCSs and their equivalent functional weights (Wi)

Category	BCS	Structure	w_i	RTPA notation
Sequence	Sequence (SEQ)		1	$P \rightarrow Q$
Branch	If-then-[else] (ITE)		2	$(? \text{exp} \text{BL} = \text{T}) \rightarrow P$ $ (\text{?} \rightarrow) \rightarrow Q$
	Case (CASE)		3	$? \text{exp} \text{BL} =$ $0 \rightarrow P_0$ $ 1 \rightarrow P_1$ $ \dots$ $ n-1 \rightarrow P_{n-1}$ $ \text{else} \rightarrow \emptyset$
Iteration	For-do (R _i)		3	$\sum_{i=1}^n R_{i-1}$
	Repeat-until (R ₊)		3	$\text{exp} \text{BL} \star \text{T}$ $R_{\geq 1} (P)$
	While-do (R ₀)		3	$\text{exp} \text{BL} \star \text{T}$ $R_{\geq 0} (P)$
Embedded Component	Function call (FC)		2	$P \hookrightarrow F$ (Note: Only consider user defined functions.)
	Recursion (REC)		3	$P \cup P$
Concurrency	Parallel (PAR)		4	$P \parallel Q$
	Interrupt (INT)		4	P $\parallel \text{circ}(\text{circ} \text{S} \nearrow Q \searrow \text{circ})$

Definition 10. The *equivalent functional size* (EFS) of a software component, S_f , is defined as a product that is proportional to its number of inputs (N_i), number of outputs (N_o), and the sum of structural weights of its n BCSs (W_{bcs}), i.e.:

$$S_f = N_i \times N_o \times W_{bcs} = N_i \times N_o \times \sum_{k=1}^n W_{bcs}(k) \text{ [EFU]} \quad (9)$$

where the unit of EFS is the equivalent functional unit (EFU) as defined in Formula 8.

The EFU provides a physical meaning for the functional size of software for the first time [19, 20], indicating that one EFU of software is the simplest component that consists of only unity input, output and equivalent functional weight of BCS. This is a new development beyond conventional function points as a measure of software functionality.

Based on Formula 9, the EFS of a complex component with n_m methods, $S_f(C)$, can be derived as follows:

$$S_f(C) = \sum_{i=1}^{n_m} S_{f-m}(i) \text{ [EFU]} \quad (10)$$

where $S_{f-m}(i)$ is the EFS of the i th method that can be directly measured according to Formula 9.

Similarly, the EFS of a component-based system \hat{S}_f , is obtained below:

$$\hat{S}_f = \sum_{j=1}^{n_c} S_f(C_j) = \sum_{j=1}^{n_c} \sum_{i=1}^{n_m} S_{f-m}(j, i) \text{ [EFU]} \quad (11)$$

where n_c is the number of components in the software system.

Example 1: A simple software component, *MaxFinder*, is given in Figure 2. The equivalent weights for each of the three BCSs can be determined according to the definitions in Table 1.

```

Program MaxFinder,

max, x: int;

begin
  max := 0;           // BCS1(sequence): W1 = 1
  read (x);
  while x<>0 do      // BCS2(iteration): W2 = 3
  begin
    if x > max       // BCS3(branch): W3 = 2
    then max := x;
    read (x);
  end;
  write (max);
end.

```

Figure 2. Example of a software component

For the given example above, it can be determined that $N_i = 1$, $N_o = 1$, and $W_{bcs} = \sum_{i=1}^3 W_i = 1+3+2 = 6$. Thus, the EFS of this component can be derived as:

$$S_f = N_i \times N_o \times W_{bcs} = 1 \times 1 \times 6 = 6 \text{ [EFU]}$$

It is noteworthy in Figure 2 that only one sequential structure is considered for an entire method.

Based on the measures of physical size (S_p) and function size (S_f) developed respectively in Sections 3 and 4, an interesting relationship between them, the *code functional efficiency* (e_f), can be derived as shown below:

$$e_f = S_f / S_p \text{ [EFU/LOC]} \quad (12)$$

It is found if only the measure of physical sizes is adopted in software engineering, programmers would intend to develop larger software than necessary, know as “the *fatware*.” The code functional efficiency as described in Formula 12 can be introduced to deal with this tendency. Because the equivalent functional size (S_f) of software is stable for a given software component or system, the large the physical size of the implementation (S_p), the lower the coding efficiency (e_f). Therefore, Formula 12 provides a useful measure for controlling implementation efficiency of software projects.

5. MEASUREMENT OF COMPLEXITY OF SOFTWARE COMPONENTS

There are various approaches toward the measurement of software complexity [11, 24]. Conventionally, *Structural complexity* of software is measured by the McCabe cyclomatic metric [15], while *functional complexity* of software is measured by function points [2]. As developed in Section 4, EFS provides a new approach to measure both the functional and structural complexities of software components in a coherent way, and takes advantages of both approaches.

5.1 Measuring component complexity by McCabe's cyclomatic metric

When a component or system is represented by a control flow graph (CFG), the *complexity of a component*, $O'(C)$, can be measured by using the McCabe *cyclomatic complexity* metric [15, 16].

Definition 11. The *cyclomatic complexity* of a given component C , $O'(C)$, is defined by the following formula:

$$O'(C) = v(G) = e - n + 2p \text{ [MCI]} \quad (13)$$

where e is the number of edges in the CFG of the component, representing branches and cycles; n the number of nodes in the CFG that is equivalent to a set of continuous sequential code; and p the number of connected components in the CFG, where, for a single given component, $p=1$. The unit of $O'(C)$ is a *McCabe cyclomatic index* (MCI).

The McCabe cyclomatic complexity measure is a variation of Euler's theorem in graph theory [13] defined as following:

$$n - e + r = 2 \quad (13')$$

where r is the number of regions, or the bounded areas + 1, in a connected graph. Comparing Formula 13' and Formula 13, it can be seen that MCI is equivalent to the number of regions, r , in Euler's theorem. It is noteworthy that Euler's theorem, or Formula 13', holds only for a connected graph. Therefore, when $p > 1$, neither Formula 13' or 13 is applicable.

Henderson-Sellers and his colleague suggested that the $2p$ section in Formula 13 should be $1+p$ [10], although the numerical results would be the same when $p=1$, i.e.:

$$O'(C) = e - n + p + 1 \text{ [MCI]} \quad (13'')$$

If we consider a disconnected graph be separated components with $p=1$ for each of them, then Formula 13'' is always equivalent to Formula 13.

Definition 12. The *architectural complexity of a component-based system*, $O'(\hat{S})$, is defined as the sum of the McCabe cyclomatic complexities of its n_c components enclosed in the software system, i.e.:

$$O'(\hat{S}) = \sum_{j=1}^{n_c} O'(C_j) \text{ [MCI]} \quad (14)$$

Example 2. For a software component as shown in Figure 2, its CFG can be derived as shown in Figure 3. According to Formula 13, its McCabe cyclomatic complexity is:

$$O'(C) = e - n + 2p = 7 - 6 + 2 * 1 = 3 \text{ [MCI]}$$

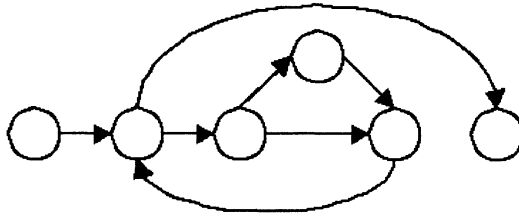


Figure 3. A derived flow-graph for Example 1

5.2 Measuring component complexity by EFS

It is noteworthy that in the McCabe cyclomatic approach as described in Section 5.1, only internal structures (loops and branches) are considered for measuring a component's architectural complexity, while I/Os that may significantly affect functional complexity of a component are not considered. This problem motivates the investigation of a new approach to measure the complexity of components that takes into account both internal structures and I/Os.

The new approach to determining the architectural and functional complexities of component-based software is the EFS-based method as developed in Section 4. The EFS-based complexities of component $O(C)$ and system $O(\hat{S})$ can be defined respectively as shown below:

$$O(C) = \sum_{i=1}^{n_m} S_f(i) \text{ [EFU]} \quad (15)$$

$$O(\hat{S}) = \sum_{j=1}^{n_c} O(C) = \sum_{j=1}^{n_c} \sum_{i=1}^{n_m} S_f(j, i) \text{ [EFU]} \quad (16)$$

Example 3. Taking the same example as shown in Figure 2 and Example 1, the component complexity of this component is:

$$O(C) = S_f(C) = 6 \text{ [EFU]}$$

The result shows that by considering both the architectural and functional factors, component's complexity is equivalent to six EFUs.

6. MEASUREMENT OF ARCHITECTURAL ATTRIBUTES OF SOFTWARE COMPONENTS

Supplementing the functional and complexity measures as developed in Sections 3 to 5, this section describes a set of common measures for software architectural attributes. Emphasis will be put on cohesion and coupling, which provide useful indications of the design quality of component-based software systems [5].

Definition 13. *Class fan-out, FO_{cl}* , is defined as the number of immediate successor classes that directly inherit from a given class in the inheritance hierarchy, i.e.:

$FO_{cl} = \#(\text{immediate successor classes})$ (17) Formula 17 is equivalent to the number of children metric in CKM [7].

Definition 14. *Depth of inheritance, DP_{ih}* , is defined as the number of nodes in the longest branch of the inheritance tree from the root to the leaf class.

$DP_{ih} = \max \{\#(\text{nodes of inheritance branches})\}$ (18) This formula is equivalent to the *depth of inheritance tree metric, DIT* , in CKM [7].

Definition 15. *Level of class reuse, $L_{cl-reuse}$* , is defined as a ratio between the number of inherited classes (and objects) n'_{cl} and the total number of classes (and objects) in an OO software n_{cl} , i.e.:

$$L_{cl-reuse} = (n'_{cl} / n_{cl}) * 100\% \quad (19)$$

This formula is equivalent to the measures proposed in [10].

In addition to the above simple architectural attributes for CBS, cohesion and coupling are a pair of advanced architectural attributes. Referring to Figure 1, the relations between a given component C_k and the remaining components in a system can be categorized into internal relations $R(C_k)$ and external relations $R'(C_k)$. The former are relations between methods that belong to C_k ; the latter are those between C_k and other components in the system [19, 20].

Definition 16. Cohesion of a component C_k , $CH(C_k)$, is defined as a ratio between the class' internal relations $R(C_k)$ and its total internal and external relations ($R'(C_k)$), i.e.:

$$CH(C_k) = \{ \#R(C_k) / (\#R(C_k) + \#R'(C_k)) \} * 100\% \quad (20)$$

where, $\#(R_{C_i})$ is the number of internal relations, and $\#(R'_{C_i})$ the number of external relations.

Definition 17. System cohesion, CH , is defined as a mathematical mean of those of the cohesion ratios of its n components in the system, i.e.:

$$CH = 1/n \sum_{k=1}^n CH(C_k) [\%] \quad (21)$$

where, $0\% \leq CH \leq 100\%$ and it is expected that the higher values are good.

Example 4. For the given component-based system as shown in Figure 1, its component and system cohesions can be determined as:

$$CH(C_1) = \{ \#(R_{C_1}) / (\#(R_{C_1}) + \#(R'_{C_1})) \} * 100\% = (3 / (3+4)) * 100\% = 42.9\%$$

$$CH(C_2) = \{ \#(R_{C_2}) / (\#(R_{C_2}) + \#(R'_{C_2})) \} * 100\% = (4 / (4+4)) * 100\% = 50.0\%$$

$$CH(C_3) = \{ \#(R_{C_3}) / (\#(R_{C_3}) + \#(R'_{C_3})) \} * 100\% = (1 / (1+2)) * 100\% = 33.3\%$$

and

$$CH = 1/n \sum_{i=1}^n CH(C_i) = 1/3 (42.9\% + 50.0\% + 33.3\%) = 42.1\%$$

Definition 18. *Coupling of a component C_k , $CP(C_k)$* , is defined as a ratio of the class' external relations $R'(C_k)$ and its total internal $R(C_k)$ and external relations, i.e.:

$$CP(C_k) = \{\#R'(C_k) / (\#R(C_k) + \#R'(C_k))\} * 100\% \quad (22)$$

Definition 19. *System coupling, CP*, is defined as a mathematical mean of all those of its n components, i.e.:

$$CP = 1/n \sum_{k=1}^n CP(C_k) [\%] \quad (23)$$

where $0\% \leq CP(C) \leq 100\%$ and lower values are better.

Example 5. For the given component-based system as shown in Figure 1, its component and system coupling are:

$$CP(C_1) = \{\#(R'_{c_1}) / (\#(R_{C_1}) + \#(R'_{c_1}))\} * 100\% = (4 / (3+4)) * 100\% = 57.1\%$$

$$CP(C_2) = \{\#(R'_{c_2}) / (\#(R_{C_2}) + \#(R'_{c_2}))\} * 100\% = (4 / (4+4)) * 100\% = 50.0\%$$

$$CP(C_3) = \{\#(R'_{c_3}) / (\#(R_{C_3}) + \#(R'_{c_3}))\} * 100\% = (2 / (1+2)) * 100\% = 66.7\%$$

And

$$CP = 1/n \sum_{i=1}^n CP(C_i) = 1/3 (57.1\% + 50.0\% + 66.7\%) = 57.9\%$$

It is noteworthy that there is an interesting supplementary relationship between a component's cohesion and coupling as follows:

$$CH(C_k) + CP(C_k) = 100\% \quad (24)$$

or, at the system level, it is:

$$CH + CP = 100\% \quad (25)$$

Therefore, when either cohesion or coupling is known, the other one can be determined easily for a component or a CBS by Formulae 24 or 25.

7. CONCLUSION

This chapter has described alternative approaches to software system measurement, especially those for component-based software. Models of component-based software have been introduced. A set of formally defined measures for the physical sizes, functional sizes, architectural/functional complexity of software components and systems have been systematically developed.

New measures on the equivalent functional size (EFS) and the physical meaning of an equivalent functional unit (EFU) have been introduced, and their advantages and applications in component-based software measurement have been demonstrated.

It has been recognized that components and component-based systems can be measured by both physical and functional sizes, and complexities of software can be determined by both architectural and functional complexity measures. This work has demonstrated that a *software measurement system* can be described in a formal and algebraic way. Thus, new measures may be derived based on existing and well-defined measures.

ACKNOWLEDGEMENTS

The author would like to acknowledge the support of the research fund of the Natural Sciences and Engineering Research Council of Canada (NSERC). The author would like to thank Prof. B. Henderson-Sellers and other reviewers for their constructive comments and suggestions.

- REFERENCES**
- [1] Abreu, F.B. and R. Carapuca (1994), Candidate Metrics for Object-Oriented Software within a Taxonomy Framework, *J. Systems and Software* 23, pp.87-96.
 - [2] Albrecht, A.J. and J.E. Gaffney (1983), Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation, *IEEE Transactions on Software Engineering*, Vol.9, No.6, pp.639-648.
 - [3] Basili, V.R. and D. Weiss (1984), A Methodology for Collecting Valid Software Engineering Data, *IEEE Trans. Software Engineering*, Vol. 10, No. 6, pp.728-738.
 - [4] Basili, V.R., C. Caldiera, H.D. Rombach (1994), Goal Question Metric Paradigm, in J.J. Marciniak ed., *Encyclopedia of Software Engineering*, Vol. 1, John Wiley & Sons, pp. 528-532.

- [5] Briand, L., J. Daly and J. Wuest, (1998), A Unified Framework for Cohesion Measurement in Object-Oriented Systems, *Empirical Software Engineering - An International Journal*, Vol.3, No.1, pp.65-117.
- [6] Bunge, M. (1967), *Scientific Research I and II*, Springer-Verlag, Berlin.
- [7] Chidamber, S.R. and C. F. Kemerer (1994), A Metrics Suite for Object Oriented Design, *IEEE Trans. on Software Engineering*, Vol. 20, pp.476-498.
- [8] Churcher, N.I. and Shepperd, M.J. (1995), Comments on "A Metrics Suite for Object-Oriented Design," *IEEE Trans. On Software Engineering*, Vol. 21, No.3, pp.263-265.
- [9] Drake, T. (1999), Metrics Used for Object-Oriented Software Quality, in S. Zamir ed., *Handbook of Object Technology*, CRC Press, Boca Raton, pp. 46.1 – 46.17.
- [10] Henderson-Sellers, B. and D. Tegarden (1994), A Critical Re-Examination of Cyclomatic Complexity Measures, *Proceedings on Software Quality and Productivity (ICSQP '94)*, pp.328-335.
- [11] Henderson-Sellers, B. (1996), *Object-Oriented Metrics – Measures of Complexity*, Prentice-Hall, Englewood Cliffs, NJ.
- [12] Hoare, C.A.R., I.J. Hayes, J. He, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Sufirin (1987), Laws of Programming, *Communications of the ACM*, Vol.30, No.8, August, pp.672-686.
- [13] Lipschutz, S, and M.L. Lipson (1997), *Schaum's Outline of Theory and Problems of Discrete Mathematics*, McGraw-Hill Co. Inc., pp. 201.
- [14] Lorenz, M. and J. Kidd (1993), *Object-Oriented Software Metrics: A Practical Guide*, Prentice-Hall ECS Professional, UK.
- [15] McCabe, T.J. (1976), A Complexity Measure, *IEEE Trans. on Software Engineering*, Vol.2, No.4, pp. 308-320.
- [16] McDermid, J. (1991), *Software Engineer's Reference Book*, Butterworth-Heinemann ltd., Oxford, UK, Chapter 30, pp. 10.
- [17] Wang, Y. and G. King (2000), *Software Engineering Processes: Principles and Applications*, CRC Press, USA, 752pp.
- [18] Wang, Y. (2001a), A Software Engineering Measurement Framework (SEMF) for Teaching Quantitative Software Engineering, *Proceedings of the 2001 Canadian Conference on Computer Engineering Education, Fredericton, Canada, May, pp. 88-101*.
- [19] Wang, Y. (2001b), Formal Description of Object-Oriented Software Measurement and Metrics in SEMS, *Proceedings of the 7th International Conference on Object-Oriented Information Systems (OOIS'01)*, Calgary, August, Canada, pp.123-132.
- [20] Wang, Y. (2002a), *Software Engineering Measurement: An Applied Framework of Software Metrics*, CRC Press, USA, to appear.
- [21] Wang, Y. (2002b), On Software Engineering Measurement Deployment in GQM, *Proceedings of the 2nd ASERC Workshop on Quantitative and Soft Computing Based Software Engineering (QSSE'02)*, Banff, AB, Canada, February, pp.1-8.
- [22] Wang, Y. (2002c), The Real-Time Process Algebra (RTPA), *Annals of Software Engineering: An International Journal*, Vol. 14, USA, Oct.
- [23] Wilson, L.B. and Clark, R.G. (1988), *Comparative Programming Languages*, Addison-Wesley Publishing Company, Wokingham, England.
- [24] Horst Zuse (1998), *A Framework of Software Measurement*, Walter de Gruyter, Berlin.

BIOGRAPHIES

Jan Øyvind Aagedal: MSc from Norwegian University of Science and Technology, 1992, and PhD from University of Oslo, 2001. PhD topic was support for quality of service (QoS) in development of distributed systems, with a focus on specification of QoS. Started in 1993 and has since 2001 been a senior scientist at SINTEF Telecom and Informatics. Has worked on CORAS since the project started in January 2001.

Colin Atkinson: Dr. Colin Atkinson is a professor at the University of Kaiserslautern, Germany, and a project leader/consultant at the affiliated Fraunhofer Institute for Experimental Software Engineering (IESE). Prior to that he was an associate professor of Software Engineering at the University of Houston - Clear Lake. His interests are centered on object and component technology and their use in the systematic development of software systems. He received a Ph.D. and M.Sc. in computer science from Imperial College, London, in 1990 and 1985 respectively, and received his B.Sc. in Mathematical Physics from the University of Nottingham in 1983.

Franck Barbier: Dr. Franck Barbier is professor in software engineering at the university of Pau (France). He is the director the computer science research institute (LIUPPA) of the university of Pau. His research interests are object modelling, component modelling, UML and seamless object/component development. He is the scientific consultant of Reich Technologies, a French company among the 17 companies that built UML 1.1 at the OMG in 1997.

Nicolas Belloir: Nicolas Belloir got his M.Sc. in computer science at University Paul Sabatier (Toulouse) in June 1999. He has worked in industry during two years for the software company Transiciel. Currently, he is PhD student (since may 2001) at the University of Pau in the LIUPPA in the AOC (Agent, Object and Component) group. His research topics are in the CBSE domain and include development methods, techniques for software composition and certification/validation of this composition.

Jean-Paul Bodeveix: he is an old student of the “Ecole Normale Supérieure” of Cachan and received a PhD of Computer Science from the University of Paris-Sud in 1989. This research was partially supported by the ESPRIT project PADMAVATI aiming at the development of a parallel architecture for symbolic computations. He is now an assistant professor of computer science at the University of Toulouse III, France. He has been interested in concurrency, typing and its connection to object-oriented programming, logic programming, rewriting techniques, formal specifications

and proofs. In this context, he has studied specification formalisms and proof environments for the validation of protocols and the semantics of development environments such as B.

Folker den Braber: Doctorandus (MSc) in Informatics at the University of Leiden, the Netherlands 2001. Thesis written about modular refinement in MSC. Employed as a research scientist at SINTEF since 2001. Interest in evolutionary algorithms, Petri-nets and theory of concurrency.

Jean-Michel Bruel: Jean-Michel Bruel got its PhD at University Paul Sabatier (Toulouse) in December 1996. Since September 1997, he is associate professor at the University of Pau. Member of the TASC laboratory from 1997 to 2000. Currently member of the LIUPPA in the Agent-Object-Component group. Its research areas include development of distributed, object-oriented applications, methods integration, and the use of formal methods for the development of distributed systems.

Agusti Canals: Agusti Canals has been working at CS SI since 1981. He is a project leader, a consultant manager and the “Club of Experts” coordinator for CS SI within the Quality and Technology Department. He has presented papers on HOOD, Ada, UML and object business patterns at several conferences (e.g. DASIA, AdaEurope, UML’Europe, ICSSEA ...), he has also published (about UML process) in the Journal Object Oriented Programming (JOOP) and he gives courses on software engineering for different training structures in Toulouse, Paris Dauphine and Strasbourg.

Juan Carlos Cruellas: he received his PhD in Telecom Engineering in 1989. His most relevant activities include participation in European projects, in standardisation groups and software developments. He has participated in, among other things, to the following European projects: TEDIS SAM Project (whose result was a new EDIFACT message, KEYMAN, devoted to certificate management); ACTS Project “MULTIMEDIATOR: Multimedia Publishing Brokerage Service” mentioned before; Telematics DEDICA project, as member of the staff at esCERT-UPC, where strongly collaborated with the technical manager; ETS PKITS project.

Theo Dimitrakos: Theo Dimitrakos is a senior scientist at CLRC Rutherford Appleton Laboratory, the UK representative at the ERCIM WG on e-commerce and a Visiting Research Fellow at King’s College, London. He has been leading CLRC participation in a number of European projects and he is actively supporting the advancement of rigorous systems engineering methods through industry and academic collaborations. Prior to his appointment at CLRC he offered Technical Consultancy to a number of UK companies and has been a systems analyst in the Electronic Commerce Group of Logica UK, plc. He has a PhD in Computing from Imperial College, UK, and a BSc in Mathematics from the University of Crete, Greece.

Petr Donth: Petr Donth was born in Czechoslovakia in 1950. He studied at the Technical University (VUT) of Brno, and graduated in 1974. He worked in the Czech research institute of water pumps (Sigma Olomouc) as a programmer and analyst in the field of computer simulation for water pumps and turbines. Seven years he worked in the Research institute of clothes (OP Prostejov) as a system analyst and project leader, mainly on large projects for clothes companies. Some years he was the Information system director in the Commerce Bank of Prague. From the year 1991 he is co-owner of company KD SOFTWARE, where he works as technical manager and project manager. He is company CEO from the year 1993.

Jose Luis Fernández-Sánchez: Associate professor at the Technical University of Madrid. His research interest includes requirements engineering, software architecture, software testing and component based development. He also works in industry where he has been involved in software development for real-time systems and component based development for avionics systems. He received a MSc in Aeronautical Engineering and a PhD in Computer Science from the Technical University of Madrid.

Rune Fredriksen: Rune Fredriksen will in short time complete his M.Sc in Computer Science at Østfold University College. The thesis is on the use of risk assessment techniques in programmable systems. Specifically, he focuses on the development of critical software in a Rational Unified Process context. In June 2001 he joined Institute For Energy Technology/OECD Halden Reactor Project where he works with safety and reliability issues in the department of Computerised Operation Support Systems.

Kurt Geihs: Kurt Geihs is a professor for Distributed Systems at the Technical University Berlin (TUB) in Germany. His research interests include distributed systems, operating systems, networks and software technology. Current projects focus on QoS management in CORBA, component-based software and middleware for mobile and ad-hoc networking. Before joining the TUB he was a professor in the Department of Computer Science at the University of Frankfurt, Germany. From 1985 - 1992 he worked for IBM at the IBM European Networking Center in Heidelberg, Germany. Prof. Geihs holds a PhD in Computer Science from the Technical University Aachen, Germany, as well as master degrees in Computer Science from the University of California, Los Angeles, California, and from the Technical University Darmstadt, Germany.

Thomas Genssler: Thomas Genssler has a diploma degree in Computer Science from the University of Dresden and is currently working on his PhD in Computer Science at the University of Karlsruhe. In addition to his academic activities, Mr. Genssler worked at an industrial consulting company

for several years and has been involved in a number of industrial consulting projects in the field of computer-aided quality management. Since 1997, Mr. Genssler has been working for the Software Engineering Department (Program Structures) at FZI (the Research Center for Computer Science) in Karlsruhe, directed by Prof. Dr. Gerhard Goos. He has participated in a number of industrial and research projects in the field of object-oriented and component-oriented software development and re-engineering. Mr. Genssler's current academic work includes research in the fields of program transformation technology and invasive software adaptation. Mr. Genssler's PhD focuses on the fields of automated software transformation and adaptation. Mr. Genssler is a regular contributor at national and international conferences on software engineering and software evolution.

Björn Axel Gran: he received his M.Sc. in industrial mathematics from the Norwegian Institute of Technology (NTNU Trondheim) in 1993, and the PhD degree in software reliability from NTNU in 2002. In 1995 he joined the OECD Halden Reactor Project, where he worked in the section on software verification and validation. His work has consisted of research within software dependability, and the main interest has been within the use of Bayesian belief networks in the safety assessment of software based systems. Since January 2001 he has been leader of the work package on Risk Analysis in the EU-funded project "CORAS" (IST-2000-25031).

Hans-Gerd Gross: Hans-Gerhard Gross received his PhD from the University of Glamorgan, UK, where he was concerned with real-time systems development and test. Currently, he is responsible for building up software testing competence at the Fraunhofer Institute for Experimental Software Engineering. His main research focus is on model-driven approaches to system development and verification.

Brian Henderson-Sellers: Brian Henderson-Sellers is Director of the Centre for Object Technology Applications and Research and professor of Information Systems at University of Technology, Sydney (UTS). He is author of eleven books on object technology and is well-known for his work in OO/CBD methodologies (MOSES, COMMA, OPEN, OOSPICE) and in OO metrics.

Siv-Hilde Houmb: Siv Hilde Houmb finished her MSc Thesis in Informatics in April 2002 on the subject Mobile E-Commerce and Stochastic Models. She worked as a system administrator and system developer at Norwegian Institute of Air Pollution (NILU) from 1997 to 1999 and at Telenor R&D from 1999 to 2002. In 2002 she joined the security and mobility group in Telenor R&D where she works with risk analysis, modelling and tiger team related activities. PhD student at the Computer and Information

Science Dept. of the Norwegian University of Science and Technology in Trondheim from August 2002 to July 2006.

Hyoseob Kim: Hyoseob Kim is a researcher in the Centre for HCI Design of City University, London, UK. His research interests include COTS evaluation and selection, software reuse, and software metrics. Currently, he is involved in the BANKSEC (SECure BANKing application assembly using a component based approach) EU Framework V Project. He received a PhD in software engineering from Durham University. He is a member of ACM.

Gerald Kotonya: Dr Gerald Kotonya is a Senior Lecturer in Software Engineering at Lancaster University's Computing Department. His principal research interests are in Component-Based Software Engineering. He is particularly interested in frameworks for developing adaptable component architectures.

Bruno Lefever: Bruno Lefever is Principal Consultant at Computer Associates. He has 15 years experience in the development of large-scale business systems and related methodologies. Since the early nineties he specialised in implementing Component-Based approaches in large IT organisations, facing the challenges to consolidate object orientation with traditional software engineering.

Neil Maiden: Neil Maiden is a Reader and Head of the Centre for Human-Computer Interface Design, an independent research department in City University's School of Informatics. He received a PhD in Computer Science from City University in 1992. He is and has been a principal and co-investigator of several EPSRC- and EU-funded research projects. His research interests include requirements engineering and component-based software engineering. Neil has over 75 journal and conference publications. He is also co-founder and treasurer of the British Computer Society Requirements Engineering Specialist Group.

Thierry Millan: He received his PhD from the Paul Sabatier University of Toulouse in 1995. Thierry MILLAN participated in two TELEMATICS projects (DEDICA, MIRTO) when he was at Alcatel. His participation in the DEDICA project allows Thierry MILLAN to have been skilled in business processes and in electronic commerce. After two years at the Alcatel Company, Thierry Millan is now a teacher of UML at the Paul Sabatier University and a researcher at the IRIT institute. His main research topics are design methodology, persistence and object-oriented languages (Java, C++, Ada 95). He has presented several papers concerning software engineering at several conferences.

Noël Plouzeau: Noël Plouzeau is an assistant professor of computer science at IRISA/IFSIC public research laboratory, Rennes, France. Since 1985 he has worked on distributed algorithms, computer supported

cooperative work (CSCW) environments, distributed applications and object-oriented design. He has led the research and development work on the Rusken distributed framework for CSCW. He is an active member of the QCCS IST project on quality controlled component based software. His current research interests focus on techniques for defining and managing non-functional properties in component based software. Noël Plouzeau is a permanent research member of the Triskell research project led by Jean-Marc Jézéquel on reliable and efficient component based software engineering.

Anne-Marie Sassen: she received in 1987 her masters degree in Computer Science from the University of Leiden (The Netherlands) and her PhD from Delft University of Technology, also in The Netherlands, in 1993. She has been working on various topics in Computer Science, among others intelligent control systems, human computer interaction, software engineering, e-learning and e-commerce. Her current position is technical coordinator of the software engineering division of SchlumberbergerSema Spain, and she is the project manager of the QCCS project.

Benedikt Schulz: he received a Diploma degree in Computer Science from the University of Karlsruhe in April 1996. He joined the Program structure group in May 1996 where he was involved in various national and European projects as researcher and later as project manager. Since February 1999 he is the head of the group. He is preparing a dissertation in the area of reengineering of object-oriented systems. Among his research interests are Object-oriented software engineering, reengineering of legacy systems, Software quality, Software processes and their improvement, component-based software engineering, aspect-oriented and adaptive programming and Mobile Computing. He has project management experience from a lot of successful projects. His experience with European Projects include the projects PECOS, TROOP, FAMOOS and IMPROVE. He is a member of the German Computer Society (GI) and member of several GI working groups including Object Technology and Reengineering.

Friedrich Stallinger: Fritz Stallinger holds a research position at the Department of Systems Engineering and Automation at the Johannes Kepler University Linz, Austria. His research interests include component-based software engineering, software process and quality management, systems thinking and system dynamics, and the application of process modelling and simulation to the areas of software engineering and software process improvement. He studied computer science at the Kepler University Linz and has been a consultant to the European automobile industry in the areas of product and strategic planning and the co-ordination of new car development projects. He is currently the project manager of the OOSPICE Project (IST-1999-29073), a European Union funded international research project

targeting at the development of a process improvement and development methodology for component-based software engineering.

Yannis C. Stamatiou: He received his PhD from the Computer Engineering & Informatics Dept. of the University of Patras in June 1998 and from September 1998 to September 1999. He was a postdoctoral fellow at the Computer Science Department of Carleton University, Ottawa, Canada under a NATO scholarship. He has extensive experience on Unix and C/C++ programming, TCP/IP programming, parallel programming as well as the design of microcontroller based secure embedded systems. He is currently senior researcher at the Computer Technology Institute and Instructor at the Department of Computer Engineering and Informatics of the University of Patras.

Ketil Stølen: PhD in formal methods from Manchester University 1990. Research fellow at Manchester University 1990-1991. Research associate at the Technical University Munich 1991-1996. Scientist at the OECD Halden Reactor Project 1996-1999. Currently senior scientist at SINTEF Telecom and Informatics (since 1999) and professor at the University of Oslo (since 1998). Has been the technical manager of CORAS since the project was started up in January 2001.

Jonathan Vincent: Dr Vincent is currently a researcher at Southampton Institute (UK) where he leads the Intelligent Systems Laboratory. He received a BEng (Hons) in Electrical and Electronic Engineering from the University of Portsmouth, and then worked for a number of years as a design engineer in embedded systems. He received an MSc in Computing (Software Engineering) and was awarded a PhD in Computer Science by Nottingham Trent University. His research interests include various aspects of software engineering and computer science, in particular, systems development methodologies, real-time and embedded systems, heuristic optimisation and parallel computation.

Yingxu Wang: Dr. Yingxu Wang is a professor of Software Engineering at the University of Calgary, Canada. He is the coordinator of “the Theoretical and Empirical Software Engineering Research Centre (TESERC)”. He received a PhD from The Nottingham Trent University / Southampton Institute, UK, and a BSc from Shanghai Tiedao University. He is the author or editor of 5 books and over 150 papers. He is the chair of IEEE ICCI’02 and program chair of OOIS’01.

Torben Weis: Torben Weis is a PhD student at the Technical University of Berlin. His research interests are in the domain of modelling QoS aware component systems with UML and the integration of QoS-contracts in distributed component systems. He holds a “Diplom-Informatiker” degree

(M.S. in Computer Science) from Goethe-University in Frankfurt, Germany, and is a core member of the KDE Open Source project.

Index

assembly, 12, 14, 18, 24, 39, 47, 50, 51, 61, 121, 122, 123, 124, 136, 141, 144, 155, 231

business component, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 27, 28, 29, 31, 36, 38, 47

commercial off-the-shelf, 2, 50, 84, 122, 159, 227, 239, 251

component integration, 51, 228, 239

component interface, 13, 14, 153, 176, 234, 235, 236

component language, 174

component modeling, 1, 8, 13, 16, 18, 19, 20, 24, 27, 28, 31

component testing, 53, 124

composability, 13, 18

connector, 18, 172, 175, 176, 177, 178, 180, 183, 186

contract, 13, 14

metrics, 247, 248

object-oriented programming, 7, 40, 45, 153, 186

quality of service, 7, 9, 12, 15, 18, 151, 152, 153, 156, 158, 161, 165, 205

requirements engineering, 1, 49, 50, 53, 54, 56, 59, 61, 124, 227, 231, 232

security, 3, 11, 12, 15, 24, 49, 57, 152, 153, 154, 159, 189, 190, 192, 193, 194, 197, 200, 201, 202, 205, 228, 233, 234, 243

software architecture, 5, 54, 186, 209, 210, 212, 220

software development
process, 5, 18, 121, 138,
152, 167, 168

standards, 2, 4, 5, 11,
12, 13, 15, 24, 52, 120,
121, 136, 137, 138, 139,
140, 144, 145, 194, 228,
239

unified modeling
language, 2, 142