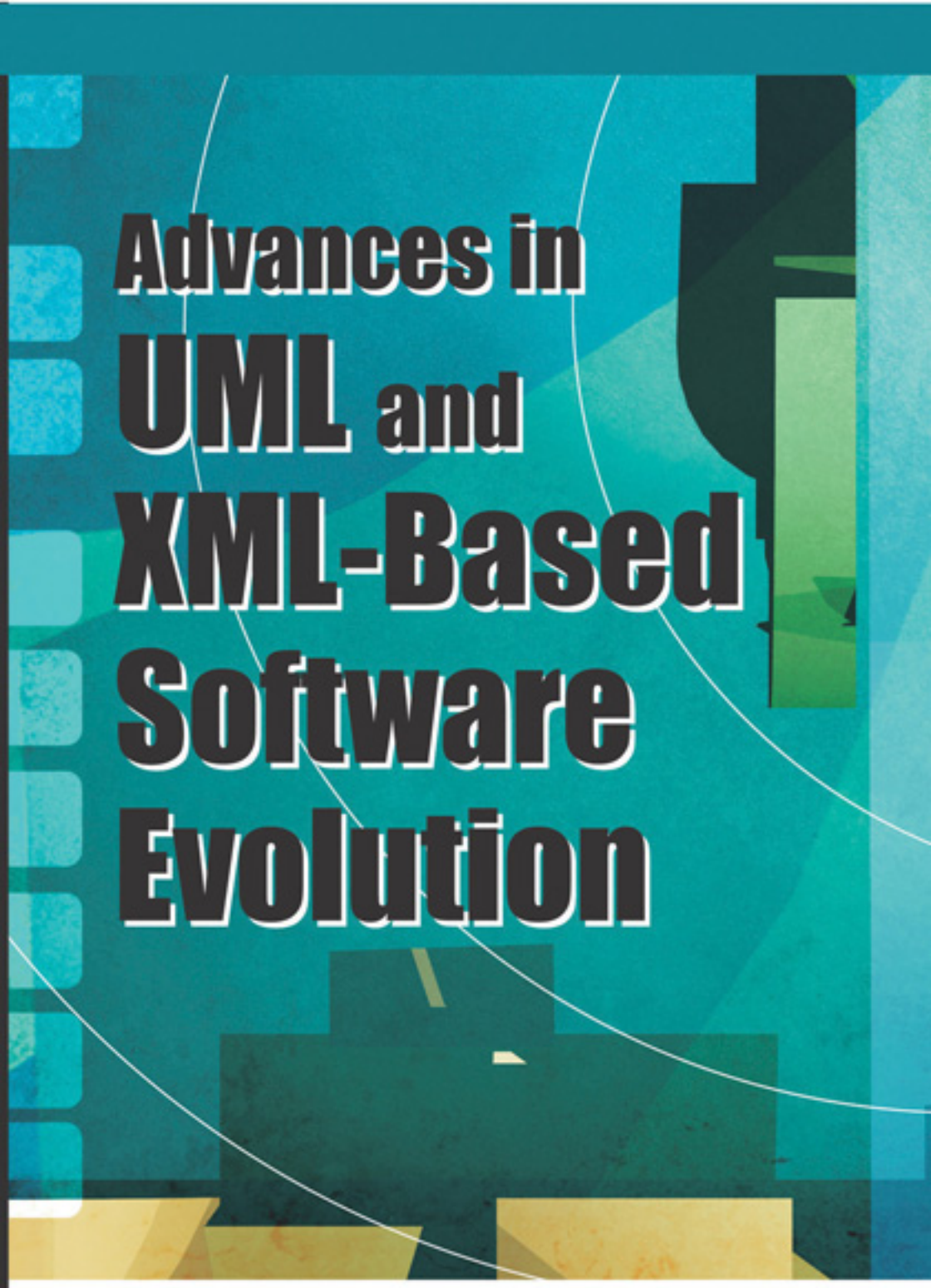


HONGJI YANG

Advances in UML and XML-Based Software Evolution

Hongji Yang
De Montfort University, UK

IDEA GROUP PUBLISHING
Hershey • London • Melbourne • Singapore

Acquisitions Editor: Renée Davies
Development Editor: Kristin Roth
Senior Managing Editor: Amanda Appicello
Managing Editor: Jennifer Neidig
Copy Editor: Amanda O'Brien
Typesetter: Cindy Consonery
Cover Design: Lisa Tosheff
Printed at: Integrated Book Technology

Published in the United States of America by
Idea Group Publishing (an imprint of Idea Group Inc.)
701 E. Chocolate Avenue, Suite 200
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@idea-group.com
Web site: http://www.idea-group.com

and in the United Kingdom by
Idea Group Publishing (an imprint of Idea Group Inc.)
3 Henrietta Street
Covent Garden
London WC2E 8LU
Tel: 44 20 7240 0856
Fax: 44 20 7379 3313
Web site: http://www.eurospan.co.uk

Library of Congress Cataloging-in-Publication Data

Advances in UML and XML-based software evolution / Hongji Yang, editor.
p. cm.
Summary: "Reports on the recent advances in UML and XML based software evolution in terms of a wider range of techniques and applications"--Provided by publisher.
Includes bibliographical references and index.
ISBN 1-59140-621-8 (hardcover) -- ISBN 1-59140-622-6 (softcover) -- ISBN 1-59140-623-4 (ebook)
1. Computer software--Development--History. 2. UML (Computer science) 3. XML (Document markup language) I. Yang, Hongji.
QA76.76.D47A378 2005
006.7'4--dc22
2005005919

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Advances in UML and XML-Based Software Evolution

Table of Contents

Preface

This book continues to provide a forum, which a recent book, *Software Evolution with UML and XML*, started, where expert insights are presented on the subject.

In that book, initial efforts were made to link together three current phenomena: software evolution, UML, and XML. In this book, focus will be on the practical side of linking them, that is, how UML and XML and their related methods/tools can assist software evolution in practice.

Considering that nowadays software starts evolving before it is delivered, an apparent feature for software evolution is that it happens *over all stages* and *over all aspects*. Therefore, all possible techniques should be explored. This book explores techniques based on UML/XML and a combination of them with other techniques (i.e., *over all techniques* from theory to tools).

Software evolution happens *at all stages*. Chapters in this book describe that software evolution issues present at stages of software architecturing, modeling/specifying, assessing, coding, validating, design recovering, program understanding, and reusing.

Software evolution happens *in all aspects*. Chapters in this book illustrate that software evolution issues are involved in Web application, embedded system, software repository, component-based development, object model, development environment, software metrics, UML use case diagram, system model, Legacy system, safety critical system, user interface, software reuse, evolution management, and variability modeling.

Software evolution needs to be facilitated *with all possible techniques*. Chapters in this book demonstrate techniques, such as formal methods, program transformation, empirical study, tool development, standardisation, visualisation, to control system changes to meet organisational and business objectives in a cost-effective way.

On the journey of the grand challenge posed by software evolution, the journey that we have to make, the contributory authors of this book have already made further advances.

Organisation of the Book

The book is organised into 15 chapters and a brief description of each chapter is as follows.

Chapter I, *Design Recovery of Web Application Transactions*, is by Scott Tilley, Damiano Distante, and Shihong Huang. Modern Web sites provide applications that are increasingly built to support the execution of business processes. In such a transaction-oriented Web site, poor transaction design may result in a system with unpredictable workflow and a lower-quality user experience. This chapter presents an example of the recovery of the "as-is" design model of a Web application transaction. The recovered design is modeled using extensions to the transaction design portion of the UML-based Ubiquitous Web Applications (UWA) framework. Recovery facilitates future evolution of the Web site.

Chapter II, *Using a Graph Transformation System to Improve the Quality Characteristics of UML-RT Specifications*, is by Lars Grunske. Architectural transformations are an appropriate technique for the development and improvement of architectural specifications. This chapter presents the concept of graph-based architecture evolution and how this concept can be applied to improve the quality characteristics of a software system, where the UML-RT used as an architectural specification language is mapped to a hypergraph-based datastructure, so that transformation operators can be specified as hypergraph transformation rules.

Chapter III, *Version Control of Software Models*, is by Marcus Alanen and Ivan Porres. Through reviewing main concepts and algorithms behind a software repository with version control capabilities for UML and other MOF-based models, this chapter discusses why source code- and XML-based repositories cannot be used to manage models and present alternative solutions that take into account specific details of MOF languages.

Chapter IV, *Support for Collaborative Component-Based Software Engineering*, is by Cornelia Boldyreff, David Nutter, Stephen Rank, Phyo Kyaw, and Janet Lavery. Collaborative system composition during design has been poorly supported by traditional CASE tools and almost exclusively focused on static composition. This chapter discusses the collaborative determination, elaboration, and evolution of design spaces that describe both static and dynamic compositions of software components from sources such as component libraries, software service directories, and reuse repositories. It also discusses the provision of cross-project global views of large software collections and historical views of individual artefacts within a collection.

Chapter V, *Migration of Persistent Object Models Using XMI*, is by Rainer Frömming and Andreas Rausch. Change is a constant reality of software development. With ever-changing customer requirements, modifications to the object model are required during software development as well as after product distribution. This chapter presents the conceptualisation and implementation of a tool for the automated migration of persistent object models. The migration is controlled by an XMI-based description of the difference between the old and the new object model.

Chapter VI, *PRAISE: A Software Development Environment to Support Software Evolution*, is by Chih-Hung Chang, William C. Chu, Chih-Wei Lu, YI-Chun Peng, and Don-

Lin Yang. This chapter first reviews current activities and studies in software standards, processes, CASE toolsets, and environments, and then proposes a process and an environment for evolution-oriented software development, known as PRocess and Agent-based Integrated Software development Environment (PRAISE). PRAISE uses an XML-based mechanism to unify various software paradigms, aiming to integrate processes, roles, toolsets, and work products to make software development more efficient.

Chapter VII, *Developing Requirements Using Use Case Modeling and the Volere Template: Establishing a Baseline for Evolution*, is by Paul Crowther. The development of a quality software product depends on a complete, consistent, and detailed requirement specification but the specification will evolve as the requirements and the environment change. This chapter provides a method of establishing the baseline in terms of the requirements of a system from which evolution metrics can be effectively applied. UML provides a series of models that can be used to develop a specification which will provide the basis of the baseline system.

Chapter VIII, *Formalizing and Analyzing UML Use Case View Using Hierarchical Predicate Transition Nets*, is by Xudong He. UML use case diagrams are used during requirements analysis to define a use case view that constitutes a system's functional model. Each use case describes a system's functionality from a user's perspective, but these use case descriptions are often informal, which are error-prone and cannot be formally analysed to detect problems in user requirements or errors introduced in system functional model. This chapter presents an approach to formally translate a use case view into a formal model in hierarchical predicate transition nets that support formal analysis.

Chapter IX, *Formal Specification of Software Model Evolution Using Contracts*, is by Claudia Pons and Gabriel Baum. During the object-oriented software development process, a variety of models of the system is built, but these models may semantically overlap. This chapter presents a classification of relationships between models along three different dimensions, proposing a formal description of them in terms of mathematical contracts.

Chapter X, *Visualising COBOL Legacy Systems with UML: An Experimental Report*, is by Steve McRobb, Richard Millham, Jianjun Pu, and Hongji Yang. This chapter presents a report of an experimental approach that uses WSL as an intermediate language for the visualisation of COBOL legacy systems in UML. Visualisation will help a software maintainer who must be able to understand the business processes being modeled by the system along with the system's functionality, structure, events, and interactions with external entities. Key UML techniques are identified that can be used for visualisation. The chapter concludes by demonstrating how this approach can be used to build a software tool that automates the visualisation task.

Chapter XI, *XML-Based Analysis of UML Models for Critical Systems Development*, is by Jan Jürjens and Pasha Shabalin. High quality development of critical systems poses serious challenges. Formal methods have not yet been used in industry as they were originally hoped. This chapter proposes to use the Unified Modeling Language (UML) as a notation together with a formally based tool-support for critical systems development. The chapter extends the UML notation with new constructs for describing criti-

cality requirements and relevant system properties, and introduces their formalisation in the context of the UML executable semantics.

Chapter XII, *Augmenting UML to Support the Design and Evolution of User Interfaces*, is by Chris Scogings and Chris Phillips. The primary focus in UML has been on support for the design and implementation of the software comprising the underlying system. Very little support is provided for the design or evolution of the user interface. This chapter first reviews UML and its support for user interface modeling, then describes Lean Cuisine+, a notation capable of modeling both dialogue structure and high-level user tasks. A case study shows that Lean Cuisine+ can be used to augment UML and provide the user interface support.

Chapter XIII, *A Reuse Definition, Assessment, and Analysis Framework for UML*, is by Donald Needham, Rodrigo Caballero, Steven Demurjian, Felix Eickhoff, and Yi Zhang. This chapter examines a formal framework for reusability assessment of development-time components and classes via metrics, refactoring guidelines, and algorithms. It argues that software engineers seeking to improve design reusability stand to benefit from tools that precisely measure the potential and actual reuse of software artefacts to achieve domain-specific reuse for an organisation's current and future products. The authors consider the reuse definition, assessment, and analysis of a UML design prior to the existence of source code, and include dependency tracking for use case and class diagrams in support of reusability analysis and refactoring for UML.

Chapter XIV, *Complexity-Based Evaluation of the Evolution of XML and UML Systems*, is by Ana Isabel Cardoso, Peter Kokol, Mitja Lenic, and Rui Gustavo Crespo. This chapter analyses current problems in the management of software evolution and argues the need to use the Chaos Theory to model software systems. Several correlation metrics are described, and the authors conclude the long-range correlation can be the most promising metrics. An industrial test case is used to illustrate that the behaviours of software evolution are represented in the Verhulst model.

Chapter XV, *Variability Expression within the Context of UML: Issues and Comparisons*, is by Patrick Tessier, Sébastien Gérard, François Terrier, and Jean-Marc Geib. Time-to-market is one severe constraint imposed on today's software engineers. This chapter presents a product line paradigm as an effective solution for managing both the variability of products and their evolutions. The product line approach calls for designing a generic and parameterised model that specifies a family of products. It is then possible to instantiate a member of that family by specialising the "parent" model or "framework," where designers explicitly model variability and commonality points among applications.

Acknowledgments

Sincerely, I would like to thank all the people who have helped with the publication of this book.

First, I would like to acknowledge the authors for their academic insights and the patience to go through the whole proposing-writing-revising-finalising process to get their chapters ready, and also for serving as reviewers to provide constructive and comprehensive reviews for chapters written by other authors.

Special thanks go to the publishing team at Idea Group Inc.; in particular, to Mehdi Khosrow-Pour whose support encouraged me to finish this continuation book in the area of software evolution with UML and XML which provides me a wonderful opportunity to work with more leading scholars in the world; to Jan Travers for her continuous support in logistics of the project; to Michele Rossi and Amanda Appicello for copyediting and typesetting the book; and to Megan Kurtz for designing the one-page promotion brochure.

Finally, I would like to thank my wife, Xiaodong, and my son, Tianxiu, for their support throughout this project.

Hongji Yang, PhD

Loughborough, UK

January 2005

Chapter I

Design Recovery of Web Application Transactions

Scott Tilley, Florida Institute of Technology, USA

Damiano Distante, University of Lecce, Italy

Shihong Huang, Florida Atlantic University, USA

Abstract

Modern Web sites provide applications that are increasingly built to support the execution of business processes. In such a transaction-oriented Web site, the user executes a series of activities in order to carry out a specific task (e.g., purchase an airplane ticket). The manner in which the activities can be executed is a consequence of the transaction design. Unfortunately, many Web sites are constructed without proper attention to transaction design. The result is a system with unpredictable workflow and a lower quality user experience. This chapter presents an example of the recovery of the "as-is" design model of a Web application transaction. The recovery procedure is prescriptive, suitable for implementation by a human subject-matter expert, possibly aided by reverse engineering technology. The recovered design is modeled using extensions to the transaction design portion of the UML-based Ubiquitous Web Applications (UWA) framework. Recovery facilitates future evolution of the Web site by making the transaction design explicit, which in turn enables engineers to make informed decisions about possible changes to the application. Design recovery of a commercial airline's Web site is used to illustrate the process.

Introduction

As with other kinds of software systems, Web sites undergo maintenance and evolve over time in response to changing circumstances. For complex Web sites supporting the execution of business processes, evolution can be particularly challenging. The hidden nature of the transaction model in the overall design of most Web sites further exacerbates the situation.

Business processes are realized as transactions that are triggered as the user executes a series of activities in order to carry out a specific task (e.g., purchase an airplane ticket). The manner in which the activities can be executed is a consequence of the transaction design. Therefore, the quality of the transaction design can have a direct influence on the quality of the user experience.

Unfortunately, many Web sites are constructed without proper attention to transaction design. It is quite common to incorrectly treat a transaction as a sequence of navigational steps through pages of the Web application (Rossi, Schmid, & Lyardet, 2003; Schmid & Rossi, 2004). The result is a system without an explicit transaction design, which leads to unpredictable workflow, maintenance difficulties, and a potentially frustrating session for the user.

This chapter presents an example of the recovery of the "as-is" design model of a Web application transaction. The recovery procedure is prescriptive, suitable for implementation by a human subject-matter expert, possibly aided by reverse engineering technology (Tilley, 2000; Müller et al., 2003). The recovered design is modeled using extensions to the transaction design portion of the Ubiquitous Web Applications (UWA) framework (UWA, 2001f). Recovery facilitates future evolution of the Web site by making the transaction design explicit, which in turn enables engineers to make informed decisions about possible changes to the application. Design recovery of a commercial airline's Web site is used to illustrate the process.

The next section outlines UWAT+, which is a refinement of the UWA transaction design model. The section "The Design Recovery Procedure" describes the design recovery procedure, including a formalization of the transactions, the creation of the Execution Model, and the construction of the Organization Model. The section "An Illustrative Example" demonstrates the use of the procedure on a representative Web site from the travel industry. Finally, "Summary" goes over the main points of the chapter and outlines possible avenues for future work.

UWAT+

The Web provides a distributed information system infrastructure as the base platform for application deployment. Indeed, one of the reasons for the success of e-commerce business today is the transactional behavior that the Web offers. However, for many Web sites that are already in use and in need of maintenance, this widely used behavior is often too complex, consisting of several ill-defined sub-transactions which can hinder

systematic evolution. The transaction design for such applications needs to be more explicit, flexible, and take users' goals into account.

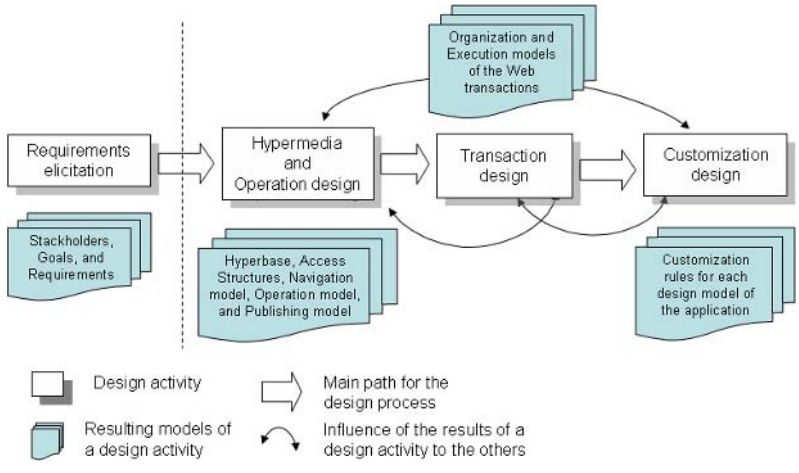
The UWA framework provides a complete design methodology for ubiquitous Web applications that are multi-channel, multi-user, and generally context-aware. As illustrated in Figure 1, the UWA design framework organizes the process of designing a Web application into four main activities (UWA, 2001a): (1) requirements elicitation (UWA, 2001b); (2) hypermedia design and operation design (UWA, 2001c); (3) transaction design (UWA, 2001d); and (4) customization design (UWA, 2001e). Each design activity results in a unique design model, which can iteratively affect the creation of other designs elsewhere in the process.

The UWA framework represents an excellent platform on which to build the conceptual modeling portion of the design recovery procedure. This section outlines a refined and extended version of the UWA framework, called UWAT+, which focuses specifically on extensions to the transaction design model. In the UWA vernacular, "transactions" represent the way business processes are addressed and implemented in Web-based applications. The extensions to the UWA transaction model include simplifications and extensions related to the definition of Activity and enhancements to several aspects of the Organization and Execution models, which are (according to the UWA) the main models on which the design of a Web transaction is based. Extensive details of UWAT+ are provided in Distante (2004); this section provides an overview of the salient features used for design recovery.

Changes to the Definition of Activity

Activities taken into account by the Organization and Execution model of a transaction implementing a business process should only be those that are meaningful for the user of the Web-based application; system-related activities and data-centered operations

Figure 1. An overview of the UWA application design process



can be de-emphasized. This implies that in UWAT+, the OperationSet of an activity is no longer considered, mainly because it is primarily related to data-level details and to the implementation of a transaction, whereas user-centered design recovery is more concerned with conceptual models.

An Activity's PropertySet is redefined to be more user-oriented, through the introduction of a new property (Suspendability), and the tuning of the semantics associated with the previously existing properties. The extended PropertySet set is now Atomicity, Consistency, Isolation, Durability, and Suspendability (ACIDS).

Changes to the Organization Model

The Organization model describes a transaction from a static point of view, modeling the hierarchical organization in terms of Activities and sub-Activities in which the Activity can be conceptually decomposed. It also describes the relations among these activities and the PropertySet of each of them. The Organization model is a particular type of Unified Modeling Language (UML) class diagram (Booch, Rumbaugh, & Jacobson, 1998), in which activities are arranged to form a tree; the main activity is represented by the root of the tree and corresponds to the entire transaction, while Activities and sub-activities are intermediate nodes and its leaves.

In UWAT+, significant changes have been made to the Organization model by dividing the possible relations between an activity A_1 and its sub-activities $A_{1.1} \ldots A_{1.n}$ into two categories: the Hierarchical Relations and the Semantic Relations. As shown in Figure 2 and Figure 3, the two categories are defined as follows:

- **Hierarchical Relations:** The set of "part-of" relations from the Organization model. It is composed of relations such as *Requires*, *RequiresOne*, and *Optional.*
- **Semantic Relations:** The set of relationships that are not a "part-of" type. Relations among sub-activities of different activities are normally part of this kind of relation. The list semantic relations currently consists of the *Visible*, *Compensates*, and *Can Use*.

The changes to the Organization model provide a better modeling instrument with which design recovery can be accomplished. In particular, the distinction between hierarchical and semantic relations permit the designer to reason about transactions in a manner that is not possible with the unadorned UWA model. This in turn can lead to improvements in support for the business processes realized by the Web application.

Changes to the Execution Model

The Execution model of a transaction defines the possible execution flow among its Activities and sub-Activities. It is a customized version of the UML Activity Diagram (Bellows, 2000), usually adopted by the software engineering community to describe

Figure 2. An example of the Organization model highlighting the hierarchical relationships

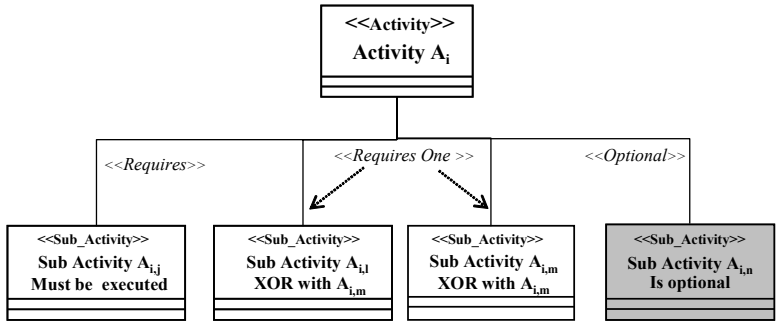
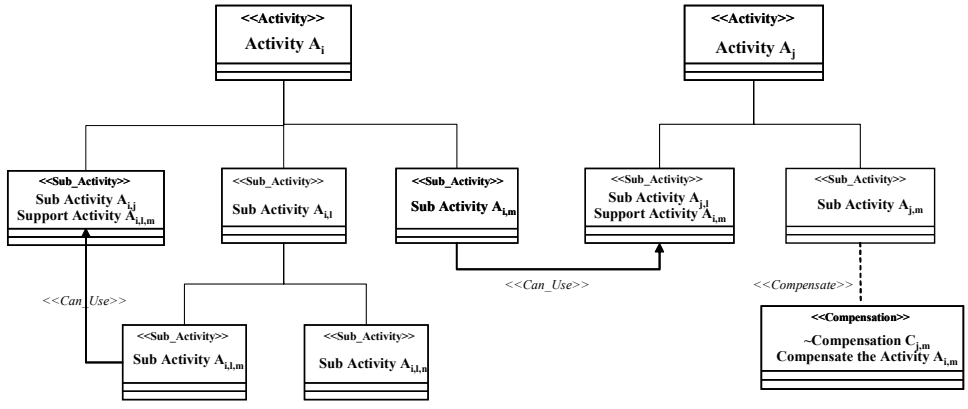


Figure 3. An example of the Organization model highlighting the semantic relationships



behavioral aspects of a system. In the execution model, the sequence of activities is described by UML Finite State Machines, Activities and sub-Activities are represented by states (ovals), and execution flow between them is represented by state transition (arcs).

The original Execution model includes both user- and system-design directions for the developer team. Since our focus is more on the former than the latter, several changes have been introduced into UWAT+.

- **Commit and Rollback Pseudo-State:** These two pseudo-states that exist in the original UWA execution model have been removed. Positive conclusion of an Activity is directly derived by the execution flow in the model, while the failure or the voluntary abort of it is modeled by the unique pseudo-state of "Process Aborted" in an Execution model.

- **Transition Between Activities:** Each possible user-permissible transition between activities must be explicitly represented in the model with a transition line between them. The actions that trigger the transition should be specified on the transition line with a transition label. Compensation activities (activities which rewind the results of others) needed to allow a transition between two activities are implicit and controlled by the system. No transition of the Execution model can be associated with the action of the user selecting the "Back" navigation button in the browser, which should be disabled in order to avoid client-side to server-side data inconsistencies.
- **Transition Labels:** A classification of the possible labels that can be associated with the transition lines of an Execution model has been introduced, with a simple labeling mechanism being used to indicate the category of the transition:
  - A: Action invoked by the user;
  - C: Condition(s) required for Activity execution;
  - R: Result of activity execution; and
  - S: State associated with system due to Activity.
- **Failure Causes and Actions Table**: A list of causes of Activity failure and possible actions the user or the system can take is maintained. The list also explains why an Activity fails and how the user or the system can react.
- **Adoption of Swimlanes**: It is suggested that swimlane diagrams (OMG, 2003) be adopted when it is useful to describe how two or more user-types of the application collaborate in the execution and completion of a transaction.

The changes to the Execution model provide better visibility into the dynamic execution paths the user will experience while completing a specific transaction. By making such paths explicit, improvements in the transaction design can be more easily accomplished. However, for such paths to be modeled properly for existing Web sites, they must first be recovered.

The Design Recovery Procedure

Given an existing Web site, the goal is to populate an instance of the UWAT+ model described in the previous section with data from the site's content and structure. The resultant model can then be used to guide restructuring decisions based on objective information concerning the quality attributes of the business process' implementation by the Web-based application. The model can be recreated using a three-step prescriptive design recovery procedure: (1) formalization of the transactions; (2) creation of the Execution model; and (3) construction of the Organization model for each of the identified transactions.

A human subject-matter expert can accomplish this design recovery procedure without any tool support. However, as described in the subsection "Future Work" at the end of this chapter, the use of automated reverse engineering technology (Chikofsky & Cross, 1990) may improve the efficacy of the process. Extensive details of the design recovery procedure using reverse engineering are provided in (Distante, Parveen, & Tilley, 2004); this section provides an overview of these three steps.

Formalization of the Transactions

In the first step of the process, the user-types of the application and their main goals/tasks are formalized. Only goals/tasks that can be defined as "operative" are considered and a transaction is associated with each of them. Overlapping tasks of two or more user-types suggests UML swimlanes in the corresponding transaction's Execution model. At the end of this step the list of transactions implemented by the application is obtained.

Creation of the Execution Model

For each of the transactions found in the first step, the Execution model is created by first performing a high-level analysis of the transaction in order to gain a basic understanding of its component Activities and Execution Flow. The transaction is then characterized as "simple" (linear) or "composite" (with two or more alternative execution paths). If the transaction is composite, then it should be further decomposed into sub-transactions until only simple transactions remain. Each simple transaction can be investigated separately. To each transaction (simple and composite) an Activity of the Execution model (and later of the Organization model) is associated.

A first draft of the Execution model is created for each simple transaction identified by executing it in a straightforward manner. Failure events are not yet taken into account in the model. The draft Execution model is then refined with deeper analysis of the transaction. All the operations available to the user during the execution of the transaction are invoked. Erroneous or incomplete data are provided in order to model failure states and possible actions the user can undertake. In this analysis phase, new secondary execution flows of the transaction can be found, and the reverse modeling technique could be invoked recursively as needed.

Finally, the table that describes the possible failure causes and the corresponding user actions or system invocations is investigated for each of the sub-activities that have been found.

Construction of the Organization Model

Once the Execution model has been obtained for a transaction, the Organization model can be constructed, which will model the transaction from a static point of view. The Execution model is used to determine the set of Activities and sub-Activities of a

transaction. In the case of a simple transaction, the set is determined by all the Activities and sub-Activities encountered in the single flow of execution available to the user. In the case of a composite transaction, the set is composed of the union of the Activities and sub-Activities of the single transaction that have been found for the composite transaction.

The tree structure of the Organization model is constructed by aggregating sub-Activities that are conceptually part of an Activity ancestor. Singleton Activities that are related to the transaction are modeled with a Can-Use Semantic Relation. Each arc in the tree represents either a hierarchical or semantic relation. To define Hierarchical Relations, the analyst can refer to the Execution Flow defined by the Execution model and conditions and execution rules defined in it. However, defining the semantic relations still requires direct inspection of the application.

For each Activity and sub-activity, it is necessary to define the value for the ACIDS PropertySet. The analyst is required to refer to the definition given for each of the properties in the UWA documentation and discover the value to be assigned to each of them through direct inspection using the Web-based application.

An Illustrative Example

To illustrate the potential benefits of design recovery, this section of the chapter focuses on the use of the procedure described in the previous section. This technique is used to recover the as-is transaction design model using the formalism outlined in the section "UWAT+" of a real-world Web-based application. The application selected is the flight reservation system of Alitalia airlines (Alitalia, 2004). The Italian Alitalia Web site (*www.alitalia.it*) was chosen because it is representative of a commonly used e-commerce application, and one that appears to offer significant room for improvement from a user's perspective. The analysis refers to a period of observation from November to December 2003. It should be emphasized that it was the Italian version of the Alitalia Web site that was analyzed; the versions for other locales, such as the USA, have been found to be quite different.

The specific transaction from the Web site used to illustrate the design recovery procedure is called "*Round-Trip Flight Reservation.*" The next two subsections describe the Organization and Execution models representing this transaction recovered from the Alitalia Web site. The subsection "Discussion" narrates some of the perceived shortcomings of the recovered transaction design that become apparent using these two models.

The recovery was realized using a manual reverse engineering process. There is no inherent reason why this process could not be made more efficient through the use of appropriate tool support. For example, one possible useful tool would be a UML editor with UWAT+ profiles. However, such tools do not as yet exist.

The Recovered Organization Model

The recovered "as-is" Organization model of the "*Complete Flight Reservation*" process is shown in Figure 4. The Organization model has a tree structure, with the Activity corresponding to its root representing the main process. The model includes the Hierarchical and Semantic Relations existing among the Activities, and for each Activity, the PropertySet it verifies.

The Activity names used in the model are purposely lengthy, so that they indicate some of the characteristics worthy of attention later in the recovery process. The user-type simulated in the analysis is "*Nonregistered User.*" Unless otherwise stated, this is the user-type implied in the following discussion.

A registered user can choose to execute one of the following three activities to reserve a flight using the Alitalia Web site: *Fast Flight Reservation*, *Complete Flight Reservation*, and *Managing Reserved Flights*. These activities are in fact connected to the main activity of the diagram (the reservation process) with a *Requires One* relation. This last activity is available only to the user type *Registered User*. The *Payment* activity is an optional activity that the users could execute if they wish to purchase the corresponding ticket online. The hierarchical relation of *Optional* that links it to its ancestor indicates this.

The model in Figure 4 details the *Complete Flight Reservation* Activity; the other Activities (which are represented by a filled version of the UML Class Stereotype), are omitted for lack of space. The PropertySet of the *Complete Flight Reservation* activity is set to *AID* (*Atomic*, *Isolated,* and *Durable*). It was observed that the user could experience inconsistency among data visualized, so the Activity is not *Consistent*. Moreover, the Activity is not *Suspendable* because it cannot be suspended in any time; instead, it must be completed during one usage session of the application.

Figure 4. The "as-is" Organization model of the "Complete Flight Reservation" Activityin the Alitalia.it Web site

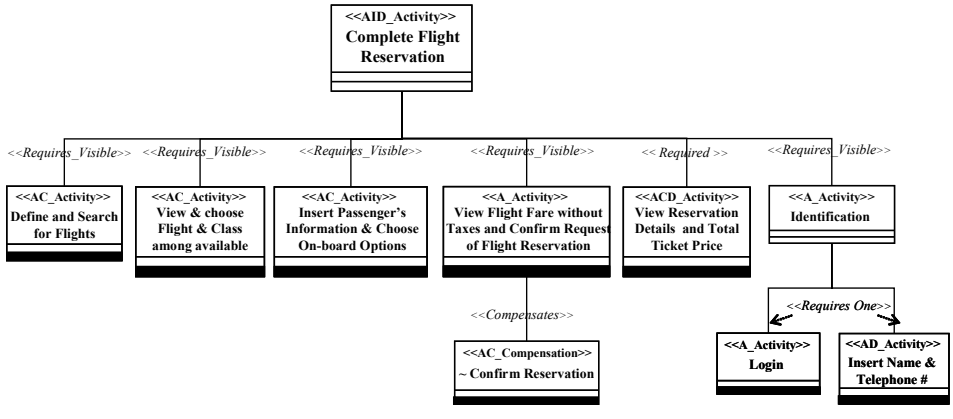
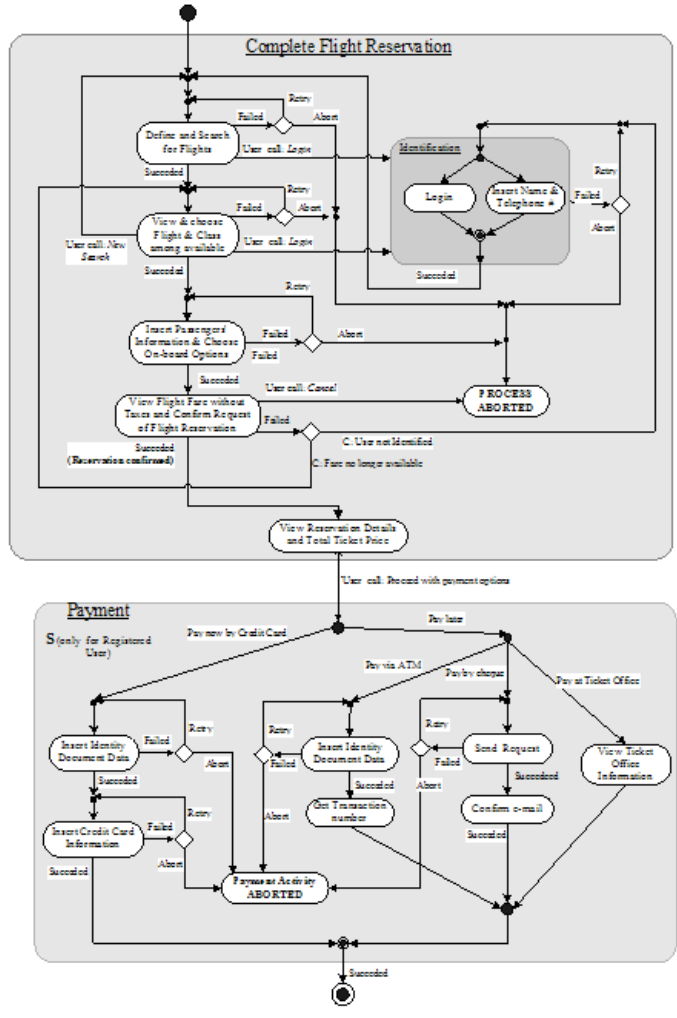


Figure 5. The "as-is" Execution model of the "Complete Flight Reservation" Activity in the Alitalia.it Web site



The diagram also shows the sub-activities into which the *Complete Flight Reservation* Activity can be conceptually decomposed. These are *Define and Search for Flights, Choose Flight & Class Among available, Insert Passenger's Information & Choose On-board Options, View Flight Fare Without Taxes and Confirm Request of Flight Reservation, View Reservation Details and Total Ticket Price,* and *Identification.* All of these are activities required for the main *Complete Flight Reservation* Activity to be completed.

The activities that correspond to the leaves of the tree are elementary ones the user normally executes in a single episode. The model shows that the user can accomplish the *Identification* Activity by either logging into the system (*Login* activity, available only for registered users) or providing a name and a telephone number *(Insert Name and*

Figure 6. The activity of "Define and Search for Flights" in the Alitalia.it Web site

Telephone # activity). These two activities are in fact related to their ancestor with a *Requires One* hierarchical relation.

Most of the activities in the recovered Organization model were found to be *Visible* since the changes they affect on data are visible by other activities during a session. The *~Confirm Reservation* activity is a *Compensates* activity which in essence rewards the effects of a successfully completed reservation when the user decides to delete it.

The Recovered Execution Model

The recovered "as-is" Execution model of the *Complete Flight Reservation* process is shown in Figure 5. The model details the activities of *Complete Flight Reservation* and *Payment* of the Organization model described in the "UWAT+" section. In the following discussion, the most linear flow of execution the user can experience while reserving a seat using the Alitalia Web site is used. (Note that while the Execution model also depicts the sub-Activities that compose the *Payment* Activity and describes the set of payment options the user can choose from, this activity is quite simply structured and is therefore not the focus of the design recovery process.)

The process of *Round-Trip Flight Reservation* requires five steps to be completed. These steps are illustrated in sequence by referring to the Execution model in Figure 5 and the screenshots of the Web page of the application supporting each of the five activities.

Step 1. The user starts the flight reservation process by defining the request of "*Round-Trip Flight Reservation*" and starting the flight search (*Define and Search for Flights*).

Figure 7. The activity of "Choose Path & Class Among Available" in the Alitalia.it Web site

Figure 6 shows a screenshot of the Alitalia Web page for this Activity. For this Activity to be successfully executed, the user must indicate the number of passengers, the preferred class, the departure and destination airports, and the departure and return dates. Default values are provided for most of the required input parameters; utilities that might have been helpful, such as a calendar, are not available to the user.

Step 2. If the flight search is successful, a list of possible flights (with different routes between departure and destination, and different times) is proposed for the itinerary (with an indication of the traveling class) specified in the previous step. The user is then required to choose the preferred path from the departure airport to the destination, and class of travel (*View & Choose Flight & Class Among Available*). Even if the preferred traveling class was specified in Step 1 of the process, the system still shows flights belonging to other classes. Figure 7 shows the screenshot of the Web page that enables the user to execute this Activity.

Step 3. To proceed ahead toward the completion of the reservation process, the user is required to provide all the information about the traveling passengers and choose for them the On-Board options such as the preferred meal and eventual needs for assistance services (*Insert Passenger's Information & Choose On-board Options*). Figure 8 shows a screenshot of the Web page in charge of this activity.

Step 4. After the previous activities have been successfully completed in sequence, the flight details and fare (without taxes) are shown. The user is asked to confirm the choices in order to effectively request the flight reservation and the system to commit it (*View Flight Fare Without Taxes and Confirm Request of Flight Reservation).* This is shown in Figure 9.

Figure 8. The "Insert Passenger Information & Choose On-board Options" Activity

Figure 9. The "View Fare Without Taxes and Confirm Request of Flight Reservation" Activity

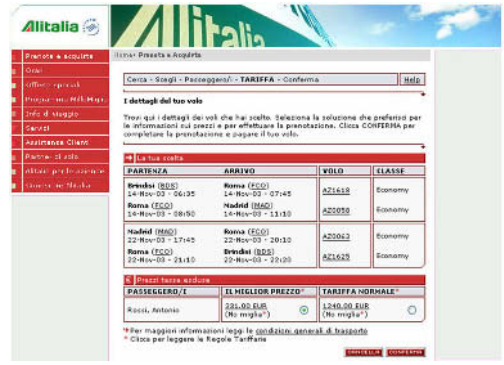
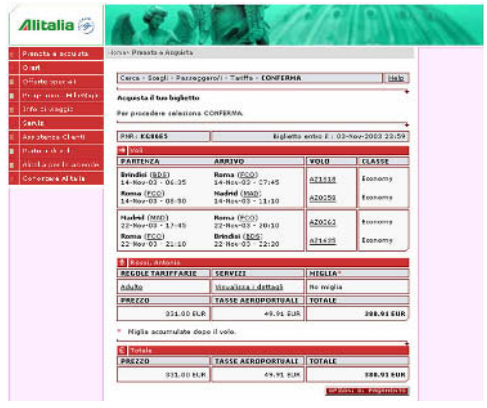


Figure 10. The "View Reservation Details and Total Ticket Price with Taxes" Activity



Step 5. If the flight reservation request succeeded, then the activity of *Complete Flight Reservation,* represented in Figure 5 by a large rounded rectangle as background, can be considered successfully completed. As shown in Figure 10, at this point the user is provided with the flight reservation code, the date they must purchase the ticket, details of the reserved flight, and the total price (taxes included) (*View Reservation Details and Total Ticket Price*). A necessary condition for the reservation confirmation to be successful is that the user has previously been identified to the system by executing the *Identification* Activity. This is accomplished by either logging into the system (in the case of a registered user), or by specifying first and last name and providing a telephone number (in the case of an unregistered user).

When Step 5 is completed, the user can exit the process or proceed with the *Payment* for purchasing the ticket corresponding to the reservation just confirmed by the system. However, only the link that starts the *Payment* activity is made explicit by the application; neither "Exit" nor "End Process" button is provided to the user to indicate that the reservation process has been completed and can be exited. The model shows that only the user call "*Proceed with Payment Options*" is provided to the user. This may be considered a navigation problem, but if known at design time, it could have been avoided.

In the case of a registered user, the reservation is stored in Personal Travel Book and from here the user can pay the ticket online at a later time. In the case of an unregistered user, if the process is exited without contextually executing the *Payment* activity (e.g., simply closing the browser or navigating outside the pages related with the process), the user has no way to restart the Activity and to buy the ticket in a later session. The previous consideration tells us that the *Payment* activity is *Suspendable* only for registered users.

As mentioned at the beginning of this section, Steps 1 to 5 depict the normal, linear execution flow for the process of *Complete Flight Reservation*. However, depending on the user's choices and the system's responses, several execution paths can be followed. For example, the execution flow can take a different path from the one described if one of the involved activities fails.

Failure causes and corresponding actions that can be undertaken by the user or the system in response to them are described by *Failure Tables*. Two of these are summarized in Table 1 and Table 2. The first table refers to the activity of *Define and Search for Flights,* while the second refers to the possible failure causes identified for the activity of *View Flight Fare Without Taxes and Confirm Request of Flight Reservation*.

Discussion

A number of observations can be made regarding possible areas of improvement for the Alitalia transaction design, based on the as-is recovered Organization and Execution models and taking into account other information extracted from the application during its direct analysis. Five areas that can be considered the most important are: the

Table 1. Failure table for Activity "Define and Search for Flights"

Cause of Failure	Application Behavior	Possible Actions	
		Action	Triggered by
A mandatory field is not inserted.	The application informs the user to fill the field.	Retry Abort	User User
The user specifies the same date for the departure and return flights in an international itinerary.	The application informs the user that return date trip cannot be the same of the departure.	Retry Abort	User User
The user asks for a flight with the same departure and destination.	The system informs the user of the error.	Retry Abort	User User

Table 2. Failure table for Activity "View Flight Fare Without Taxes and Confirm Request of Flight Reservation"

Cause of Failure	Application Behavior	Possible Actions	
		Action	Triggered by
The user does not choose any suggested flight.	The application informs the user that he must choose one of the suggested flights.	Retry Abort	User User
The system cannot find a flight in the space of ±3 days since the specified day.	The system informs the user that any flight cannot be found and invites the user to retry.	Retry Abort	User User
The flight is not available for a number of reasons	The application redirects the user to complete search page.	Abort	System

suspendability of the entire process of *Complete Flight Reservation*, the *Identification* Activity, the *Insert Passenger's Information & Choose On-board Options* activity, visualizing total cost, and restarting the process. Each of these shortcomings (SC) is discussed in turn.

SC1: Suspendability of the Entire Process

The first consideration concerns the *Suspendability* of the entire "*Complete Flight Reservation*" transaction. As shown by the Execution model, none of the activities involved in the process are *Suspendable*. For example, the user cannot store a flight they found interesting to continue the reservation process in a following session.

In addition, the *Payment* activity, as noted in Step 5 of the previous section, is *Suspendable* for registered users only. An unregistered user has no chance to purchase the ticket in a following usage session of the application. One could argue that this is an implementation issue. One could also argue that this is a matter of security policy. In either case, the designer could document the tradeoffs regarding design decisions such as *Suspendability* if the rationale were made explicit in the model.

SC2: The Identification Activity

The second important consideration is related to the *Identification* Activity. As shown by the Organization model in Figure 4, the *Identification* activity is required for the successful completion of the "*Complete Flight Reservation*" process. In this case, the shortcomings lie in the way the user is forced to access and execute this activity and what its execution causes.The user can start the *Identification* Activity with the user call *Login* (following a link provided by the application) only when executing the *Define and Search for Flights* or the *Choose Flight & Class Among Available* activities. Moreover, as described by the transition line of the Execution model in Figure 5, executing the

Identification activity causes a loss of the state of the transaction (implementation of the process) and forces the user to restart the process.This also happens when the user reaches the confirmation step (Step 5) of the reservation process and, because the user did not do it before, the activity fails in requesting the user to identify themselves.

SC3: The Insert Passenger's Information & Choose On-Board Options

Another aspect worthy of attention is the *Insert Passenger's Information & Choose On-board Options* Activity. As shown by the Execution model in Figure 5 and described in the previous section (Step 3), this Activity is required in order to carry out the reservation process. The reverse modeling process exposed the fact that this activity is executed before the user knows the fare of the selected flight (Step 4), and each time the user conducts a new search or selection of flights. It rapidly becomes very frustrating to the user to have to repeatedly input the same information. Iteratively searching for flights by changing dates and itineraries looking for the best fare available is a very common activity in any airline flight reservation application. Rather than acting as it does now, the Alitalia system should instead request the Passenger's Information only one time, and then display the fare of the chosen flight during the entire session of usage. The fact that the system loses the information entered by the user when executing this activity is also modeled by the lack of the *Durability* property in the Organization model in Figure 4.

SC4: Visualizing Total Cost

The fourth issue regards the visualization of the total cost, including taxes, of the chosen flight. Users can easily find themselves guessing between available flights shown by the system in Step 2, but needing to reach Step 4 to see the flight fare. Moreover, the price shown in this Step, as the names of the Activity suggests, does not include taxes, and only after confirming the reservation request will the user finally know the total ticket price.

SC5: Restarting the Process

Last but not least is the consideration about how much easier it is for the user to start over with another search for flights looking for a better fare. Since this is a common goal for most of the users, the application should offer the opportunity to start a new search at nearly every point of the reservation process. Instead, as the Execution model in Figure 5 shows, the application explicitly provides a way to start a new search only at Step 2, which is prior to the user knowing the price of a chosen flight. The user can attempt to get around this limitation by using the "Back" button in the browser, but this invites problems related to session expiration.

Summary

This chapter presented an example of design recovery of Web application transactions. The design recovery procedure relies on UWAT+, which is a conceptual model that is based on extensions to the transaction design portion of the UWA framework. The prescriptive recovery procedure is composed of three steps, which can be accomplished by a human subject-matter expert. A commercial airline's flight reservation system was used to illustrate the procedure.To use the procedure, there is a one-time learning curve required for the engineer to become familiar with the UWA design framework and the UWAT+ refinements. However, once this is done, the procedure is relatively easy to understand and systematic to apply. The designer is provided with a sequence of clear steps to be carried out and a set of well-defined concepts to refer to, represented by means of the well-known notation of UML diagrams. Applying the reverse modeling technique allows the analyst to draw from the application and effectively represent with the models a lot of information perceived by the user and worthy of attention from their point of view.The design recovery procedure provides the analyst/designer with a tool (broadly intended and not specifically a software tool) that is able to represent most of the aspects related to the user execution of a process to carry out their goals. UWAT+ relies on two models, the Organization model and Execution models, that, taken together, are suitable for describing at a conceptual level the design of Web application transactions. These are strong bases of discussion and comparison of ideas and strategies that the Web application realizes.

Future Work

One area of future work we foresee is to develop tool support for the design recovery procedure. Such tool support would greatly improve the likelihood of adoption by easing the reverse modeling task. It would also make the analysis phase faster and more thorough than the current manual approach. Supporting tools could range from commercial UML diagramming editors, provided with UWAT+ Organization and Execution model profiles (plug-in), to semi-automatic tailored tools able to analyze and model the Activities of an identified transaction.Another area of future work is to use the recovered design as a guide to reengineering the Web site's transactions. The following three steps outline a possible reengineering technique for Web application transactions:

1. Perform the transaction design recovery of the Web site using the procedure described in this chapter;
2. Analyze the recovered "as-is" UWAT+ transaction design model and evaluate it according to quality attributes such as usability and fulfillment of business requirements; and

3. Develop a new version of the transaction via restructuring of the as-is model, resulting in a candidate "to-be" model that better meets the user's expectations and improves the user's experiences using the Web site.

Evidence-based techniques such as empirical studies could be used to verify that the resultant Web site is "better" in some quantifiable way than the original. The difficulty is in quantifying "better," both for the designer and the developer. For the designer, one measure might be shorter time-to-market for a complex Web application, while still retaining and even improving functionality and lower subsequent maintenance costs. For the user, the measure is likely to remain usability—something that is notoriously difficult to measure, but ultimately the most important attribute of all for any application.

Acknowledgments

Tauhida Parveen contributed to the development of an early draft of this chapter.

References

Alitalia (2003). Available online at *www.alitalia.it*

Bellows, J. (2000). Activity diagrams and operation architecture. *CBD-HQ White paper*. Available online at *www.cbd-hq.com*

Booch, G., Rumbaugh, J., & Jacobson, I. (1998). *The unified modeling language user guide,* (Rational Corporation Software). Reading, MA: Addison-Wesley.

Chikofsy, E. & Cross, J. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software, 7*(1), 13-17.

Distante, D. (2004). *Reengineering legacy applications and web transactions: An extended version of the UWA transaction design model.* Unpublished Doctoral Dissertation, University of Lecce, Italy.

Distante, D., Parveen, T. & Tilley, S. (2004, June 24-26). Towards a technique for reverse engineering web transactions from a user's perspective. In *Proceedings of the 12th International Workshop on Program Comprehension*, *IWPC 2004*, Bari, Italy, (pp. 142-150). Los Alamitos, CA: IEEE Computer Society Press.

Müller, H., Jahnke, J., Smith, D., Storey, M.-A., Tilley, S., & Wong, K. (2003). Reverse engineering: A roadmap. In A. Finkelstein (Ed.), *The future of software engineering* (pp. 47-60). New York: ACM Press.

Object Management Group (OMG) (2003). *Unified language modeling specification, version 1.5.* Available online at *www.omg.org*

Rossi, G., Schmid, H. & Lyardet, F. (2003). Engineering business processes in web applications: modeling and navigation issues. *Proceedings of the 3rd International Workshop on Web Oriented Software Technology (IWWOST 2003),* Oviedo, Spain.

Schmid, H. & Rossi, G. (2004). Modeling and designing processes in e-commerce applications. *IEEE Internet Computing*, January/February.

Tilley, S. (2000). The canonical activities of reverse engineering. *Annals of Software Engineering, 9,* (pp. 249-271). Dordrecht, The Netherlands: Baltzer Scientific / Kluwer Academic.

UWA (Ubiquitous Web Applications) Project (2001a). *Deliverable D3: Requirements investigation for Bank121 pilot application.* Available online at *www.uwaproject.org*

UWA (Ubiquitous Web Applications) Project (2001b). *Deliverable D6: Requirements elicitation: model, notation and tool architecture.* Available online at *www.uwaproject.org*

UWA (Ubiquitous Web Applications) Project (2001c). *Deliverable D7: hypermedia and operation design: Model and tool architecture.* Available online at *www.uwaproject.org*

UWA (Ubiquitous Web Applications) Project (2001d). *Deliverable D8: Transaction design.* Available online at *www.uwaproject.org*

UWA (Ubiquitous Web Applications) Project (2001e). *Deliverable D9: Customization design model, notation and tool architecture.* Available online at *www.uwaproject.org*

UWA (Ubiquitous Web Applications) Project (2001f). *The UWA approach to modeling ubiquitous Web application.* Available online at *www.uwaproject.org*

Chapter II

Using a Graph Transformation System to Improve the Quality Characteristics of UML-RT Specifications

Lars Grunske, University of Potsdam, Germany

Abstract

This chapter presents the concept of graph-based architecture evolution and how this concept can be applied to improve the quality characteristics of a software system. For this purpose, the UML-RT used as an architectural specification language is mapped to a hypergraph-based data structure. Thus, transformation operators can be specified as hypergraph transformation rules and applied automatically.

Introduction

Over the past few years, software intensive technical or embedded systems have increasingly been implemented in software components (Douglas, 1999; Gomaa, 2000; Liggesmeyer, 2000). These software components have to fulfill requirements relating to quality characteristics or nonfunctional properties (NFPs), such as safety, availability, reliability, and temporal correctness. If a system does not fulfill these requirements, the

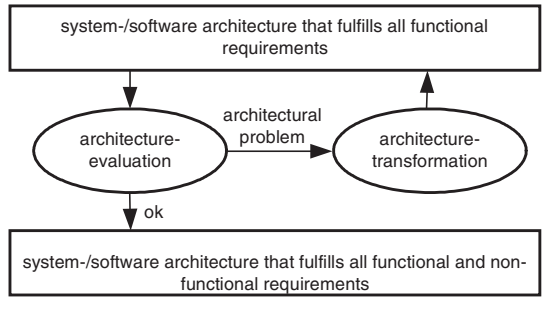
system must be restructured to improve the quality characteristics. Due to economical reasons, this change must be made as early as possible, preferably in the design phase, after the development of the system/software architecture. Based on the architecture specification, for the first time, an evaluation of the quality characteristics of the system is possible.

For the restructuring of software architectures in Bosch and Molin (1999), a cyclic process (see Figure 1) is presented that can be used to improve the quality characteristics of software architectures. The precondition for its application is an architectural specification that fulfills all functional requirements. Based on this specification, the quality characteristics are determined by an evaluation of the architecture. If the architectural specification does not meet its quality requirements, the software architecture must be restructured by the application of transformation operators. These transformation operators should influence the quality characteristics without changing the functional behavior. Thus, after the transformation the architectural specification is still functionally correct. If it turns out that all quality characteristics meet their corresponding requirements, the cyclic process can be terminated and system development can proceed with the detailed design and the implementation phase.

This chapter presents the concept of hypergraph-based architectural evolution and how this concept can be applied in the process model. For this purpose UML-RT is used as an architectural description language and the relevant elements of the UML-RT metamodel are mapped to a hypergraph-based data structure. The main benefit of this approach is the possibility to model architecture transformations as hypergraph transformation rules. Consequently, this approach allows for a (semi-) automatic application. Due to the complexity of the overall setup and the precision needed, it becomes inevitable to support the evolution process with an appropriate utility. For this purpose, a tool called Balance has been developed, which provides facilities for applying the architectural transformations explained previously.

To clarify the understanding of the hypergraph-based architecture evolution, we are going to explain these items more precisely in the following sections. In the second

Figure 1. Cyclic process for the improvement of quality characteristics



section, the theoretical background of hypergraphs and hypergraph transformations is presented. The next section gives an overview of the current state-of-the-art for software evolution with graph transformation. In the fourth section, an approach on how to use graph transformation for architectural evolution is presented. The fifth section demonstrates the application of this approach in the tool balance. Finally, conclusions are drawn and directions for future work are discussed.

Theoretical Background

Hypergraphs Theory

Hypergraphs are a generalization of normal graphs, where an edge can be associated to more than two nodes. They are a data structure that can be applied in many areas (Habel, 1992).

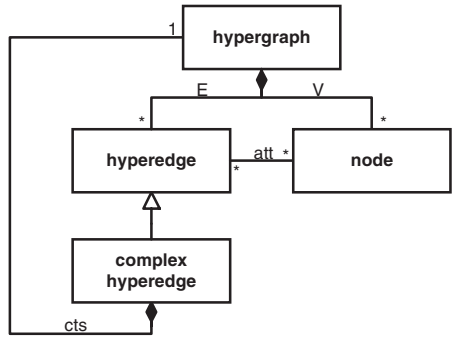
Basic Concept

Generally, a hypergraph contains a set of nodes and hyperedges. Each hyperedge can be attached to any number of nodes and each node can be attached by any number of hyperedges.

To construct a hierarchical hypergraph, we use the concept of hyperedge refinement (Drewes, Hoffmann, & Plump, 2002; Hoffmann, 2001; Hoffmann & Minas, 2001). For this purpose, special hyperedges are introduced that are used to embed another hypergraph. These hyperedges are called complex hyperedges.

Altogether, the metamodel presented in Figure 2 characterizes a typed hierarchical hypergraph.

Figure 2. Metamodel of a hierarchical typed hypergraph



Formal Definition

This section deals with the formal concept of a hierarchical typed hypergraph. Flat typed hypergraphs are introduced first, followed by multi-pointed hypergraphs. Based on these definitions, a hierarchically typed hypergraph can be defined.

According to Habel (1992) a flat typed hypergraph can be defined as follows:

Definition: Typed Hypergraph

Let L_V be a set of node types and L_E be a set of hyperedge types, then a hypergraph G from the possible set of hypergraphs G_{SET} over L_V and L_E is characterized by the tuple $<V, E, att, lab>$, with two finite sets V and E of nodes (or vertices) and hyperedges, a labeling function $lab : V \rightarrow L_V \cup E \rightarrow L_E$ and an attachment function $att : E \cup V^*$, where V^* denotes a sequence of nodes with a specified order.

The labeling function allocates for each hyperedge, and each node, a hyperedge or node type. The attachment function *att* assigns a sequence of nodes to a hyperedge. The number of elements in this sequence $|att(e)|$ is called the arity of the hyperedge.

Hierarchical hypergraphs are introduced in Drewes et al. (2002) and Hoffmann (2001) by the refinement of special hyperedges. These hyperedges are used to embed a multi-pointed hypergraph, which contains external nodes and is defined as follows:

Definition: Multi-Pointed Hypergraph

A multi-pointed hypergraph is characterized by the tuple $<V, E, att, lab, ext>$, where $<V, E, att, lab>$ is a typed hypergraph and *ext* describes a sequence of external nodes $ext \in V^*$.

The arity of a multi-pointed hypergraph can be determined by the length of the external node sequence $|ext|$. For the embedding of a multi-pointed hypergraph into another graph a hyperedge with the same arity can be refined. For this reason, the hyperedge must be removed from the graph and the multi-pointed hypergraph included in the remaining hyperedge frame. This hyperedge frame consists only of the associated nodes of the removed edge. The nodes of the hyperedge frame and the external nodes are mapped and define the glue between the enclosing graph and the embedded hypergraph (Drewes et al., 2002).

Based on the hyperedge refinement and the definition of a multi-pointed hypergraph, a hierarchical typed hypergraph can be defined as follows:

Definition: Hierarchical Typed Hypergraph

A hierarchical typed hypergraph G from the set of hypergraphs G_{SET} over L_V and L_E is characterized by the tuple $<V, E, att, lab, ext, cts>$, where $<V, E, att, lab, ext>$ is a multi-pointed hypergraph and $cts : E \rightarrow G_{SET}$ is an assignment function which assigns contained hierarchical typed hypergraphs to a hyperedge.

Due to the recursive nature of this definition, the structure of a hierarchical hypergraph is defined inductively over levels of the hierarchy. In the lowest level G^0_{SET}, no hyperedge

$e \in E$ contains an embedded hypergraph $cts(e) = \varnothing$. Thus, all hypergraphs in G^0_{SET} are regular typed graphs. In higher hierarchy levels $G^{\geq 1}_{SET}$ with the function $cts : E \rightarrow G_{SET}$ the embedded graphs are assigned to the hyperedges.

Based on object-oriented concepts, the node types L_V and hyperedge types L_E can be specified as classes. These classes can contain a set of application-specific attributes and operations. Furthermore, hypergraphs can be typed as classes.

Specific classes can be derived by inheritance from the classes of the metamodel. These classes extend the set of attributes and operations. In addition, with the introduction of classes it is easy to define variables and integrate them into a hypergraph specification. These variables can be instantiated from the class itself or from a subclass.

Hypergraph Transformation

Basic Principles

Before graph transformation rules can be specified, some basic principles have to be introduced. One of them is the identification of a subhypergraph that can be defined as follows (Habel, 1992):

Definition: Subhypergraph

Let $G = <V, E, att, lab>$ and $G' = <V', E', att', lab'>$ be two hypergraphs. Then G' is called a subhypergraph of G, denoted by $G' \subseteq G$, if $V' \subseteq V$, $E' \subseteq E$ and $att(e) \subseteq att'(e)$, $lab(e) = lab'(e)$ $lab(v) = lab'(v)$ for all edges $e \in E'$ and nodes $v \in V'$.

A subhypergraph specifies the exact occurrence in an application graph—a graph to which a transformation is to be applied. This exact identification restricts the application of a transformation rule. Thus, the identification of a subhypergraph is done by a hypergraph morphism, which is a structure- and type-preserving mapping (Habel, 1992):

Definition: Hypergraph Morphism

Let $G = <V, E, att, lab>$ and $G' = <V', E', att', lab'>$ be two hypergraphs. A hypergraph morphism $m : G \rightarrow G'$ consists of a pair of mappings $<m_V, m_E>$, with $m_V : V \rightarrow V'$ and $m_E : E \rightarrow E'$ which satisfy the following conditions:

$$\forall e \in E : lab'\left(m_E(e)\right) = lab(e)$$

$$\forall v \in V : lab'\left(m_v(v)\right) = lab(v)$$

$$\forall e \in E : att'\left(m_E(e)\right) = m_V^*\left(att(e)\right)$$

If both mappings $m_V : V \rightarrow V'$ and $m_E : E \rightarrow E'$ are injective (surjective, bijective), then the mapping $m : G \rightarrow G'$ is injective (surjective, bijective). Furthermore, if the mapping

$m : G \rightarrow G'$ is bijective, the morphism is called an isomorphism and both graphs G and G' are isomorphic denoted by $G \cong G'$.

In addition to the identification of a subhypergraph in an application graph for the graph transformation, it is necessary to define how a subhypergraph can be removed and added to an application graph. The hypergraph addition will be constructed by a disjoint union of the node and hyperedge sets. The removal of a subhypergraph can be constructed separately by the difference of sets for nodes and hyperedges.

Definition: Disjoint Union of Hypergraphs, Hypergraph Addition

Let $G = <V, E, att, lab>$ and $G' = <V', E', att', lab'>$ be two hypergraphs with $V \cap V' = \varnothing$ and $E \cap E' = \varnothing$, then the disjoint union $G+G'$ is defined by the tuple $<V \cup V', E \cup E', att_{G+G'}, lab_{G+G'}>$, with $att_{G+G'}$ and $lab_{G+G'}$ which can be constructed as follows:

$$att_{G+G'}(e) = \begin{cases} att_{G+G'}(e) = att(e) & if\ e \in E \\ att_{G+G'}(e) = att'(e) & otherwise \end{cases}$$

$$lab_{G+G'}(a) = \begin{cases} lab_{G+G'}(a) = lab(a) & if\ a \in E \cup V \\ lab_{G+G'}(a) = lab'(a) & otherwise \end{cases}$$

Definition: Removal of a Subhypergraph, Hypergraph Subtraction

Let $G = <V, E, att, lab>$ and $G' = <V', E', att', lab'>$ be two hypergraphs, with $G' \subseteq G$, then the hypergraph $G - G'$ is characterized by the tuple $<V - V', E - E', att_{G-G'}, lab_{G-G'}>$, where:

$$att_{G-G'}(e) = att(e) \doteq (V'), \forall e \in (E - E')$$
$$lab_{G-G'}(a) = lab(a), \forall a \in (V - V') \cup (E - E')$$

Hypergraph Replacement

A hypergraph transformation rule defines in an abstract manner the replacement of a subhypergraph by another in an application graph. A hypergraph transformation rule can be formally defined as follows (Corradini et al., 1997; Ehrig, 1979):

Definition: Hypergraph Transformation Rule

A hypergraph transformation rule is a tuple $<G_L, G_I, G_R, l, r>$, with three hypergraphs G_L, G_R, G_I, $\in G$ called left-hand-side graph, right-hand-side graph and interface graph and two hypergraph morphisms $l : G_I \rightarrow G_L$ and $r : G_I \rightarrow G_R$. For simplification, a hypergraph transformation rule can be denoted with $G_L \leftarrow G_I \rightarrow G_R$.

For the application of a hypergraph transformation rule, the following algorithm can be used:

- Find an occurrence morphism $o : G_L \rightarrow G$ that identifies the left-hand-side graph G_L in an application graph G.
- Check the following two conditions:
  - Dangling condition: No hyperedge $e \in E_G - E_{o(G_L)}$ is associated with a node $k \in V_{o(G_L)} - V_{o(G_I)}$
  - Identification condition: If two hyperedges $x, y \in E_{G_L}$ or nodes $x, y \in V_{G_L}$ are identified by o with $o(x) = o(y)$, then these hyperedges or nodes are elements of the interface graph $x, y \in E_{G_I} \cup V_{G_I}$
- Remove the occurrence of the left-hand-side hypergraph except for the interface graph from the application graph. The resulting graph is called context graph $D = G - o(G_L - G_I)$.
- Add the right-hand-side G_R except for G_I to the context graph resulting in $G' = D+(G_R - G_I)$ and connect all hyperedges $e \in E_{G_R} - E_{G_I}$ which are associated to a node $k \in V_{G_I}$ to the corresponding node of the context graph $o'(k)$, where $o' : G_I \rightarrow D$.

The dangling condition and the identification condition must be checked to get a syntactically correct graph after the application of the graph transformation rule. If the dangling condition fails, hyperedges which are associated to already removed nodes exist in the context graph. If the identification condition is neglected, elements exist in the application graph which must be simultaneously preserved and removed. Thus, the context graph cannot be constructed. Due to simplicity, for the implementation of a hypergraph transformation rule the algebraic approach (Ehrig, 1979; Corradini et al., 1997) is preferred. This approach is based on the construction of pushouts and pushout complements in the category of typed graphs. To visualize a graph transformation rule and its application within the algebraic approach the following pushout diagram can be used:

$$\begin{array}{ccccc} G_L & \overset{l}{\leftarrow} & G_I & \overset{r}{\rightarrow} & G_R \\ \downarrow^{o} & & \downarrow^{o'} & & \downarrow^{o''} \\ G & \leftarrow & D & \rightarrow & G' \end{array}$$

The context graph D and the morphism o' are constructed by pushout complements of the tuple $<G_L, o, l>$. Subsequently, the resulting graph G' can be constructed with the *pushout* of the tuple $<G_I, o, r>$.

Type-Generic Graph Transformation

Type-generic graph transformation improves the expressiveness of graph transformation rules by allowing the identification of subtypes with the occurrence morphism $o : G_L \rightarrow G$ (Mens, 1999). For this reason, the partial order for the subtype relationship must be defined first.

Definition: Partial Ordered Type Hierarchies

A type hierarchy over the nodetypes L_V, edgetypes L_E and hypergraph types $G \in G_{SET}$ can be defined by partially ordered relations $\sqsubseteq_V$, $\sqsubseteq_E$, $\sqsubseteq_G$, where $x \sqsubseteq y$ means that the type x is a subtype of y.

Based on this, a subtype preserving hypergraph morphism can be defined as follows.

Definition: Subtype Preserving Hypergraph Morphism

Let $G = <V, E, att, lab>$ and $G' = <V', E', att', lab'>$ be two hypergraphs. A hypergraph morphism $m : G \rightarrow G'$ consists of a pair of mappings $<m_V, m_E>$, with $m_V : V \rightarrow V'$ and $m_E : E \rightarrow E'$ which satisfy the following conditions:

$$\forall e \in E : lab'\left(m_E(e)\right) \sqsubseteq_E lab(e)$$

$$\forall v \in V : lab'\left(m_V(v)\right) \sqsubseteq_V lab(v)$$

$$\forall e \in E : att'\left(m_E(e)\right) = m_V^*\left(att(e)\right)$$

This subtype preserving hypergraph morphism allows the algebraic specification (Ehrig, 1979) of a type-generic graph transformation rule as presented in Mens (1999).

Hypergraph Transformation in Hierarchical Typed Hypergraphs

For the hypergraph replacement in hierarchical typed hypergraphs, the morphisms must first be introduced into this category. This can be defined according to Drews, Hoffmann, and Plump (2002) and Hoffmann and Minas (2001) inductively over the embedding hierarchies:

Definition: Morphism in Hierarchical Typed Hypergraphs

Let $G = <V, E, att, lab, cts>$ and $G' = <V', E', att', lab', cts'>$ be two hierarchical typed hypergraphs, then a morphism $m : G \rightarrow G'$ is defined by a tuple $<m_V, m_E, M>$, where $<m_V, m_E>$ characterizes a morphism of a flat graph and M is a family of morphisms M_e for the embedded hypergraphs of all complex hyperedges $e \in \text{dom}(cts)$. Thereby, each morphism M_e can be defined as follows:

$$M_e : cts(e) \rightarrow cts'(m_E(e))$$

Drews, Hoffmann, and Plump (2002) show that in the category of hierarchical typed hypergraphs, pushouts and pushout complements can be constructed. Thus, the application of a hypergraph transformation rule can be applied similar to the algebraic approach (Ehrig 1979).

Graph-Based Software Evolution: An Overview

In this section, the existing approaches for graph-based transformation of software specifications will be reviewed. For this purpose a short introduction into relevant approaches is given. The goals, the theoretical background, and the practical realization of each approach are presented. At the end of this section all approaches are compared in Table 1 in order to decide which concepts and aspects can be used for the graph-based architecture evolution. Furthermore, the relevant aspects for qualitatively improving architectural evolution are pointed out.

Approach of T. Mens et al.

Mens (1999) introduces a general approach for the evolution and transformation of models for the object-oriented software development process. This approach aims at a consistent formalism for the evolution during the design time of the software. For this purpose, graphs and graph transformation rules are utilized, where graphs represent model-independent specification formalism, and graph transformation rules represent model-independent transformation formalism. For the specification, hierarchical typed graphs are considered in particular. This formalism is used in Mens, Demeyer, and Janssens (2002) to describe behavior-preserving transformation rules. For the specification of the transformation rules, conditional rules are used, which are graph transformation rules enhanced with preconditions. Consequently, a systematic selection of the rules is possible and conflicts with the parallel and sequential rule application can be identified. For the practical validation of the approach, a prototype realized in Fujaba is described.

Approach of G. Taentzer

Taentzer's approach (1999) describes the visual specification of the behavior and the evolution of distributed systems. In particular, the runtime-evolution of the system is considered. At this point, her approach differs from the previously presented approach.

Table 1. Software evolution with hypergraphs: An overview

Characteristics / Approaches	General Information		Specification and Transformation Notation		Process and Tool Support	
	Date of Publications	Intention of the Approaches	Specification Formalism (Special Characteristics)	Transformation Formalism (Special Characteristics)	Selection and Achievement Test	Tool Support
T. Mens et al.	99, 00, 02, 03	Development of the formal basics for the evolution of object-oriented Models	Hierarchical typed hypergraphs	Hyper graph replacement (SPO and DPO, preconditions)	Limited choice with preconditions	Prototype realized in Fujaba
G. Taentzer	99, 01	Description of a visual concept for the modeling and evolution of distributed systems	Typed graphs	Graph replacement (distributed graph transformations)		Prototype realized in AGG
D. Le Métayer	96, 98	Formalization of architecture styles by graph grammars	Hypergraphs	Proposition of Hyper edge, Hyper graph and node replacement		
H. Fahmy, R. C. Holt	98, 00, 01	Improvement of the understandability of an architecture specification	Hierarchical typed graphs +Tarski operations	Graph replacement (SPO)		Prototype realized in PROGRES
M. Wermelinger	99, 01, 02	Formalization of the dynamic architecture reconfiguration at the runtime of a system	Typed graphs	Graph replacement (DPO)		
D. Hirsch et al.	99, 00	Reconfiguration of distributed systems under retention of an architecture style	Typed hyper graphs	Hyper edge replacement (SPO)		

For the description of the runtime evolution, the concept of distributed graph transformation based on the algebraic approach (Ehrig, 1979) is used (Ehrig, Boehm, Hummert, & Löwe, 1988). The practical applicability of the approach and the application to two case studies (Taentzer, 1999) is presented by the realization of a prototype in AGG (Taentzer, Ermel, & Rudolf, 1999).

Approach of D. Le Metayer

Le Metayer (1998) describes a basic approach for the formalization of architectural styles. The idea of his approach is the specification of software architectures as a graph whose nodes describe active components and whose edges describe the communication relations between these components. Based on previous information, architecture transformations with a refinement focus can be specified as graph transformation rules. With these architecture transformations, a graph grammar can be constructed that describes a class of architectures or an architectural style.

Approach of D. Hirsch et al.

Hirsch, Montanari, and Inverardi (1999) present an approach for the architectural configuration of distributed systems. Communication systems and their basic components, such as Client, Server, Router and Bridges, are considered particularly. The goal of the reconfiguration is to refine architectures and thereby follow an architecture style. The approach resembles the work of Le Metayer (1998). Hypergraphs are used for the modeling of software architectures. Here, components are hyperedges and the commu-

nication between the components is modeled with nodes. As formalism for architecture reconfiguration context-free hyperedge replacements are used.

Approach of H. Fahmy and R. C. Holt

The goal of the approach proposed in Holt (1998) and Fahmy and Holt (2000) is to reduce the complexity and to improve the understandability of software architectures. For this purpose, simple transformations are presented which predominantly improve the coupling and binding. The transformations in Fahmy and Holt (2000) are specified as conditional graph transformation rules and are realized prototypically in PROGESS. Holt (1998) chooses a different approach for the representation of the same rules: the architecture is specified as a graph with the classic, algebraic Tarski-operators. The transformation of Holt's architecture is described by algebraic relations realized by a relation interpreter called GROK. The practical applicability of the approach is corroborated in Holt (1998) by several case studies (250-300 KLOC COBOL and C programs).

Approach of M. Wermelinger et al.

The study of Wermelinger, Lopes, and Fiadeiro (1999, 2001) aims at the formalization of the dynamic architectural reconfiguration of a system at runtime. Distributed systems are considered in particular. For the specification of architectures, an algebraic approach and the architectural description language CommUnity are utilized. The possible modifications of the architecture at runtime are described by simple transformations. An example of such transformations is the addition, removal, and refinement of components as well as the assignment of communication variables. For the application of these transformations the double-pushout approach of Ehrig (1979) is applied. Besides the simple transformations in Wermelinger et al. (2001), a possibility is presented to design complex transformations. These complex transformations use syntax constructs that are similar to simple programming languages.

Summary of the Current Approaches

The current approaches as presented in Table 1 provide a good theoretical background for the graph-based evolution of software specifications.

Nevertheless, the current approaches lack aspects that are necessary for the application of graph-based architecture evolution in the process model presented in the introduction. These aspects are:

- An architectural description language that supports the quality improvement process

- A set of transformation operators that improve quality characteristics without changing the functional properties
- A tool that supports the (semi-) automatic application of the transformation operators

To allow the utilization of graph-based architectural evolution, these aspects are discussed in the following two sections.

Quality Improving Architecture Evolution with the UML-RT

This section explains the basic concepts for the graph-based architectural evolution with UML-RT. First, an introduction to UML-RT is given. Then, a mapping of the UML-RT to a hypergraph-based data structure and an annotation with modular analysis models for the evaluation of the relevant quality characteristics are presented. Based on this, quality improving architectural transformations can be specified as hypergraph transformation rules and applied (semi-) automatically.

Introduction to UML-RT

Due to the popularity in the industrial development of embedded systems, in this approach, the UML-RT (Selic & Rumbaugh, 1998) is used as a modeling language for the structure specification of a software intensive technical system. This modeling language is based on the ROOM methodology developed by Selic, Gullekson, and Ward (1996) and is suited to model architectural specifications as presented in Cheng and Garland (2001) and Rumpe, Schoenmakers, Radermacher, and Schürr (1999).

To model architectural specifications with UML-RT, the following three principal elements are needed:

- capsules
- ports (end-ports and relay-ports)
- connectors

The *capsules* are the main entities of the architectural specification. They encapsulate the functional behavior and correspond to the ROOM concept of actors (Selic et al., 1996). Furthermore, the *capsules* are concurrent objects, which are created based on capsule-class definitions. These capsule-classes can also be defined as composites consisting

of finer, more granular *capsules*. With respect to this, the hierarchical structure between the *capsules* is defined by a (composition) hierarchy.

For communication with its environment, a *capsule* owns a set of interface objects. These interface objects are called ports and describe the functional interfaces of a *capsule*. Between the ports, point-to-point connections can be established that are used to send messages or signals. These point-to-point connections are called *connectors* and correspond to ROOM bindings (Selic & Rumbaugh, 1998). If a message is sent directly to a component, the receiving port is called an *end-port*. To communicate with a *capsule* inside a hierarchical capsule special ports are needed to forward a message from the outside of a composite *capsule* to an inner *capsule* or in the opposite direction. These ports are called *relay-ports*.

Mapping of UML-RT to Hierarchical-Typed Hypergraphs

The mapping of the principal UML-RT elements to the elements of a hierarchical typed hypergraph is presented with a type system in Figure 3. This type system contains two meta-levels. The meta-level II describes the relevant elements of a hierarchical typed hypergraph.

The meta-level I contains the metaclasses *capsule, connector*, *end-port,* and *relay-ports*, needed to model architectures in the UML-RT.

The metaclass *capsule* is derived from the metaclass *hypergraph* and the metaclasses *end-port* and *relay-ports* are derived from the metaclass *node* in the hypergraph specification. Based on this, every capsule contains a finite set of ports, because of the composition *V* in the hypergraph-meta-level. The metaclass *connector* is derived from the metaclass *hyperedge*. Consequently, a connector can connect a set of ports to model a communication connection between these ports.

Furthermore, the hypergraph-based structure specification distinguishes between flat and hierarchical capsules. A flat capsule contains only one hyperedge. This hyperedge

Figure 3. Mapping of the principal UML-RT elements to the elements of a hierarchical typed hypergraph

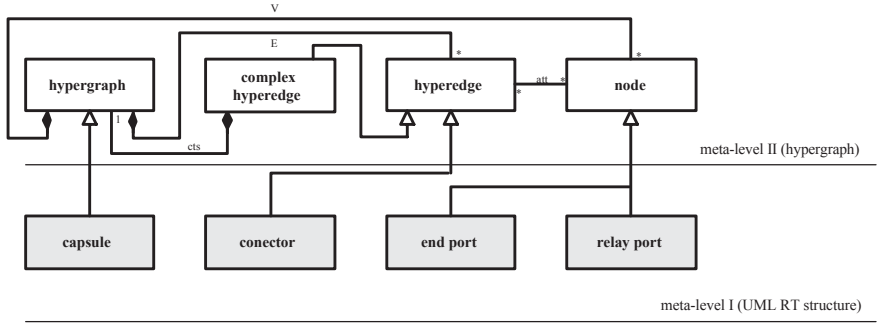
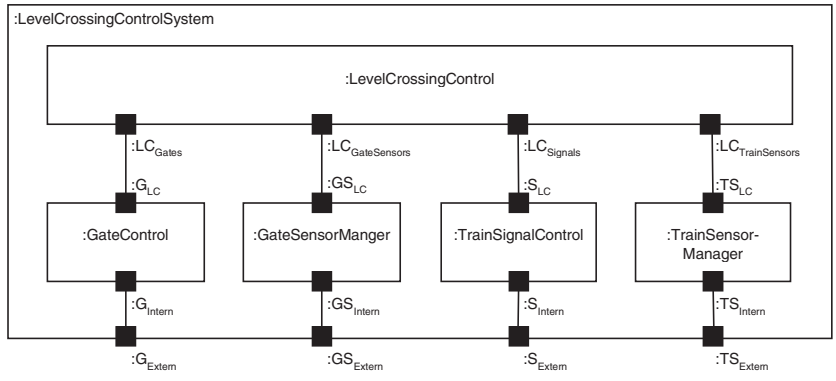


Figure 4. Structure specification of the level crossing example



is associated with all contained ports of the capsule. Hierarchical capsules can contain a set of hyperedges. Some of these hyperedges can be complex and can contain other capsules.

Level-Crossing Example

In Figure 4, a simplified version of a control system for a level crossing is modeled in UML-RT, which is specified with a capsule called *LevelCrossingControlSystem*. This capsule contains the capsules *LevelCrossingControl, GateControl, TrainSignalControl, GateSensorManager* and *TrainSensorManager*. These capsules are not further decomposed. The *LevelCrossingControl* capsule is the controller of the level crossing system. This capsule can send messages to the *GateControl* capsule to open or close the gates and to the *TrainSignalControl* capsule to allow or deny the passage for an arriving train. To get the information from the environment the *LevelCrossingControl* capsule utilizes two sensors: One sensor determines the state of the gates, and the other detects an arriving train and checks its progress through the level crossing section. These sensors are controlled by the *GateSensorManager* and *TrainSensorManager* capsules.

Due to the mapping of the principal UML-RT elements to the elements of a hierarchical typed hypergraph, as presented in the previous section, the specification of the *LevelCrossingControlSystem* capsule is a hypergraph. This hypergraph contains a node for each port. Thereby external ports are nodes of the type *relay-port* and all other ports are of the type *end-port*. Additionally, the hypergraph contains a set of hyperedges. These hyperedges are distinguished into hyperedges of the type *connector* that model the communication-connections between the ports, and of the type *hierarchical hyperedge* that contain the embedded capsules.

Table 2. Quality characteristics and relevant analysis models

Quality characteristic	Analysis model
Safety	Fault-tree-model
Reliability, maintainability, availability	Fault-tree-model and Markov chains
Temporal correctness	Scheduling models (RMA, EDFA) and End-to-End analysis
Economic attributes	Life-Cycle-Cost-model

Evaluation of Quality Characteristics

In the cyclic process presented in the introduction, software architectures must be evaluated with an architecture evaluation method to prove that the system will meet its quality requirements. For these evaluations, a set of analysis models must be integrated into the presented structural specification.

For this purpose, the analysis models presented in Table 2 are used. These models represent the state-of-the-art for each relevant quality characteristic (Birolini, 1999; Douglas, 1999; Fenelon & McDermid, 1993; Gomaa, 2000; Liggesmeyer, 2000; Pumfrey, 1999).

For the annotation of the elements of the structural specification, these analysis models are modularized as described in Papadopoulos, McDermid, Sasse, and Heiner (2001) and extend the metaclass *capsule*. An analysis model for the complete software architecture or a hierarchical capsule can be constructed with composition-based techniques according to the composition hierarchy. To apply these techniques, a modular analysis model only specifies the relevant aspects of the quality characteristics of an architectural element and can be characterized by the following parts:

- a set of outputs
- a set of inputs
- a set of internal information

The set of outputs describes the effects of the architectural element on the quality characteristics. As an example, in modular fault-trees, these outputs represent a set of failures that can be produced by the capsule and their probabilities. For the calculation of the outputs of a modular analysis model, the set of inputs and the internal information must be considered. In a fault-tree the inputs describe external failures that can influence the capsule. The internal information specifies the Boolean function that characterizes the fault tree and the probability of internal elementary failures which can be determined for an architectural element by expert knowledge or experimental studies (Birolini, 1999; Liggesmeyer, 2000; Musa, Iannino, & Okumoto, 1987).

Transformation Operators

A transformation operator describes in an abstract way the structural changes of software architectures. These changes can be:

- the removal or the addition of architectural elements
- the redirection of a connection between the architectural elements

This section gives a graphical notation and describes how to specify a graph transformation rule with this graphical notation. Thereafter, four transformation operators are presented that can be used to improve the quality characteristics of architectural specifications. A more detailed description of the transformation operators and other transformation operators can be found in Grunske (2003b).

Graphical Notation

A graphical notation is used for the representation of a transformation operator. This notation is denoted as T-notation because it groups the architectural elements of an operator into three parts, forming a T. The semantic of this notation implies that all elements or connections on the bottom-left-side of the T must be removed from the architecture. All elements or connections on the bottom-right-side of the T must be added to the architecture. The elements above the T remain unaffected and serve as gluing points between the rest of the architecture and the new added elements. This is the reason why they are redundantly contained in the upper left and the upper right side.

An example for a transformational pattern in the T-notation is given in Figure 5. This abstract operator shows that the capsules of type A and B, the two ports of type A, and the connection between them must be removed. The ports (type A, B, and C) above the T remain unaffected. They serve as gluing points for the component with type C. This component must be added to the software architecture. The application of this pattern is presented in Figure 6 for a concrete architecture.

Based on this graphical representation of the transformation operator, a graph transformation rule $TO: G^{TO}_{L} \leftarrow G^{TO}_{I} \rightarrow G^{TO}_{R}$ can be created. In this graph transformation rule the left-hand-side graph G^{TO}_{L} contains all COOL-elements in the bottom-left-side of the T. The right-hand-side graph G^{TO}_{R} contains all COOL-elements in the bottom-right-side of the T and the interface hypergraph G^{TO}_{I} is represented by the COOL-elements above the T.

Figure 5. Example of a transformational pattern in the T-notation

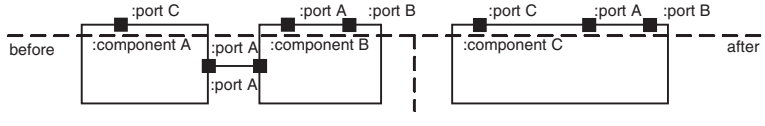
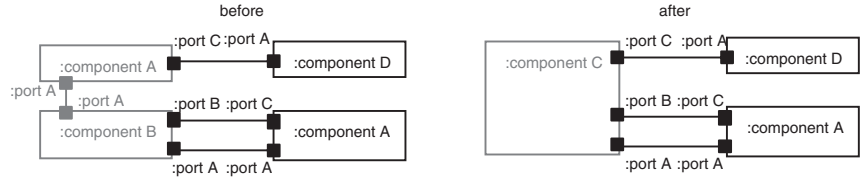


Figure 6. Example of the application of a transformational pattern



Based on this graph transformation rule, a transformation operator can be applied automatically to an architectural specification in COOL with the double pushout approach (Ehrig, 1979; Corradini et al., 1997).

Behavioral Equivalence

The automatic application of an architecture transformation operator requires a proof-algorithm to verify the behavioral equivalence before and after the application of the architectural transformation. Behavioral equivalence in this context means that the system before and after the transformation responds in the same way to each possible message trace that can be generated by the environment. Usually this is defined as trace containment or trace equivalent.

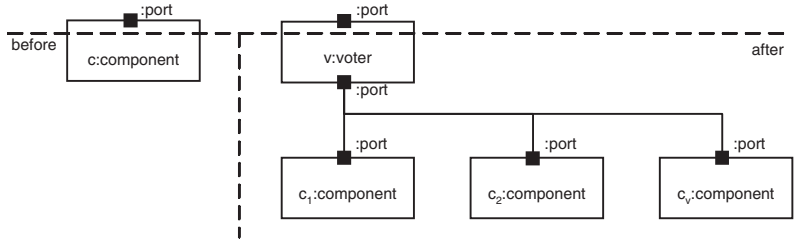
To check the behavioral equivalence a proof algorithm should contain the following steps:

1. Identify the part of the architecture that will be removed by the transformation operator
2. Identify the part of the architecture that will be added by the transformation operator
3. Construct for both parts the set of possible traces and check their equivalence

The removed part of the architecture can be constructed with the occurrence morphism $o : G_L \rightarrow G$ and the hypergraph subtraction of the interface graph G_I from the left-hand-side graph G_L as follows $o(G_L - G_I)$. The added part to the architecture $o''(G_R - G_I)$ can be similarly constructed with the morphism $o'' : G_R \rightarrow G'$.

The proof of trace equivalence of both parts is a complex problem that is extensively discussed for state charts in Harel and Kupferman (2002). For the behavioral specification with interface automata, a further proof of trace equivalence is presented in Grunske (2003a). These interface automata as proposed in de Alfaro and Henzinger (2001) are simpler than statecharts because they have no hierarchical states and no history states.

Figure 7. Transformation operator: Multi Channel Redundancy with Voting

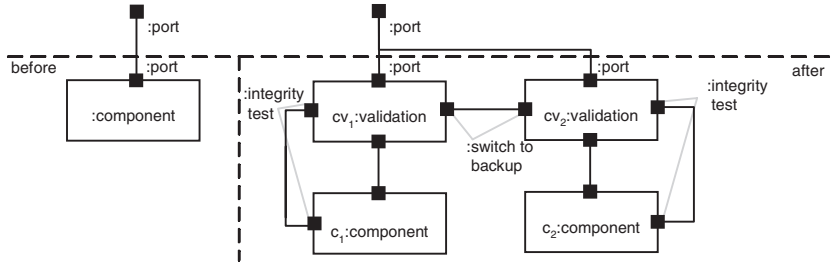


Multi Channel Redundancy with Voting

The first transformation operator we want to present is called Multi Channel Redundancy with Voting. The idea of this transformation operator is to replace a capsule that does not fulfill its safety, availability, or reliability requirements by multiple capsules on different hardware platforms and a comparator (*m-out-of-v voter*) that gets the inputs from the environment and generates multiple messages for the redundant capsules (Figure 7). Based on these messages, the capsules compute the results and send them back to the comparator, which in turn chooses the message to be sent to the environment by a majority voting. For the realization of this pattern, often three components and a two-out-of-three voting are used (Douglass, 1999, 2002). Due to the structure, random and single point failures of the hardware platforms can be detected if homogeneous software components are used. Thus, the reliability of the system will be improved if the hardware platform of the comparator is more reliable than the hardware platforms of the redundant capsules.

This operator also can be used to protect against systematic failures. Therefore, different development teams must heterogeneously implement the redundant capsules. This will reduce the probability that a systematic error in one component will affect the rest of the system (Mitra, Saxena, & McCluskey, 1999). However, Knight and Leveson (1986) point out that a diverse implementation does not detect all systematic errors, because different development teams made similar faults, and therefore the different versions do not fail independently.

Figure 8. Transformation operator: Two Channel Redundancy



Two Channel Redundancy

The transformation operator Two Channel Redundancy is similar to the transformation operator Multi Channel Redundancy with Voting. This operator replaces a capsule with two redundant channels operating in parallel on different hardware platforms (Figure 8). One channel consists of a redundant capsule of the replaced type and a capsule that is called *channel validation.* Both channels are getting all information from the environment, but one of them is active and one is passive. The active channel checks its operation with a *channel validation* capsule. The results of this validation are sent to the *channel validation* capsules of the passive channel via the connection between the *switch to backup* ports. If a failure occurs, an error is detected, or if the active channel omits to send the results, the passive channel becomes active. In this case, the former active channel must be informed. To check the correct operation of one channel the utilization of several strategies are possible (Grunske, 03b). By the application of this pattern, one channel can be still available in case of random or wear-out failures of the hardware platform of the other channel. Thus, availability can be increased by the application of this transformation operator.

Recovery Block

The transformation operator Recovery Block uses a concept developed by Randell (1975) and Randell and Xu (1995). The basic idea of this operator is to use multiple heterogeneously developed capsules operating in parallel on a single hardware platform (see Figure 9). All capsules are getting information from the environment, but one of them is the primary capsule and the others are backup capsules. The primary capsule performs the desired operations that are checked by an *acceptance test* capsule. The *acceptance test* capsule checks the primary capsule itself to detect errors. If it detects an error or a failure, the primary capsule becomes one of the backup capsules, and the next capsule in the backup pool becomes the primary capsule. To obtain protection from failures caused by systematic errors, the capsules must fail independently. For this, different development teams should implement these capsules (Avizienis, 1985).

Figure 9. Transformation operator: Recovery Block

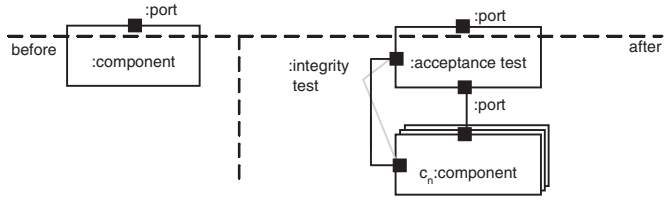
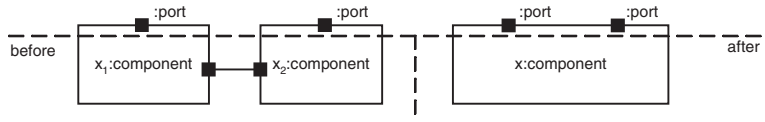


Figure 10. Transformation operator: Process Fusion



Process Fusion

The Process Fusion operator should be applied if an architecture specification contains a large number of active capsules (tasks or processes), which are processed and scheduled on a single hardware platform; the time to switch between these capsules (redirect the program counter, save and restore the registers, swap the cache pages) is high. This can have a negative effect on the temporal correctness. Therefore, the transformation operator merges two capsules into a new one, which acts like the two processes did before (see Figure 10). This will lead to scheduling, where task switching does not occur as often as before. As a result, the new components will have a better chance to meet their deadlines.

Tool Support

To support the architectural evolution process, a tool called Balance was developed. This tool is introduced next.

Short Description of Balance

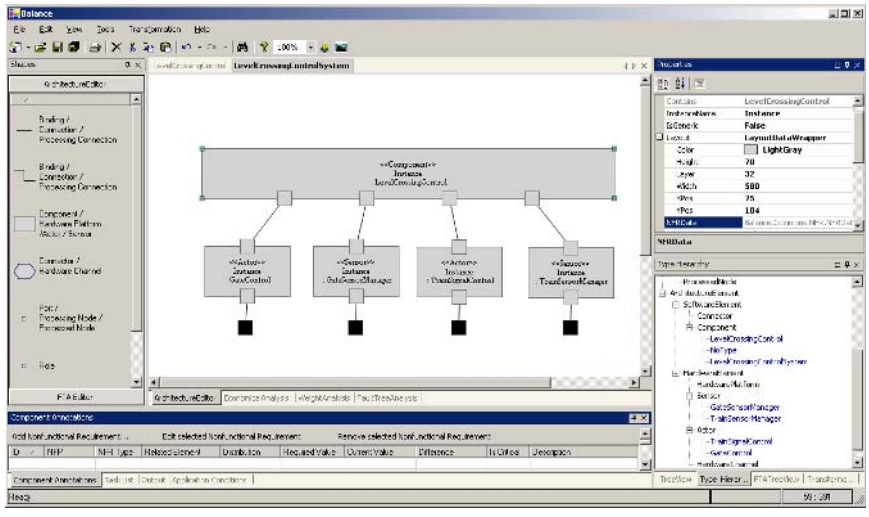
Balance can be used to model architectural specifications in UML-RT. These specifications can be directly designed by the user, or they can be extracted from a commercial tool like IBM/Rational Rose RT.

To further apply the graph-based architectural evolution process, the following steps must be taken:

- The architectural elements must be annotated with modular evaluation models for the relevant quality characteristics.
- The quality requirements must be specified.

With this information, the tool evaluates the quality characteristics of the system and proposes a set of transformation operators that will improve the architecture. The proposed transformation operators can be applied (semi-) automatically because they are specified as hypergraph transformation rules.

Figure 11. Architecture specification of the level crossing example in the tool Balance

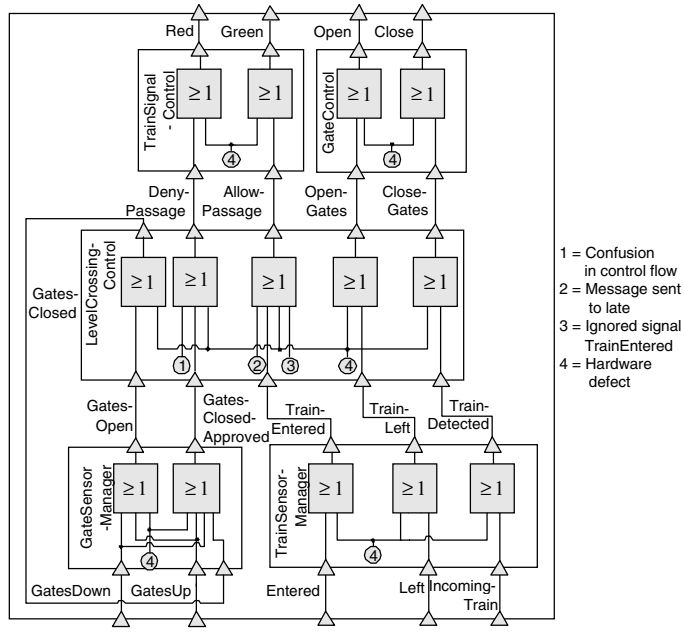


Case Study: Level Crossing Control System

The practicability of the tool is presented by modeling the level crossing control system (see Figure 11) with the objective of improving the safety properties. These safety properties are evaluated by annotating each capsule with an encapsulated fault tree. Such an encapsulated fault tree describes the failure behavior of the component. It contains a set of outputs called output failure ports defining all concrete failure types that can be caused by the component. The output failures further can be caused either by an internal fault or by an external failure of the environment or another component. The external failures that can influence the correct behavior of the component are specified with a set of input failure ports. The internal structure of an encapsulated fault tree is specified similar to normal fault trees as a Boolean function.

Starting from the structural specification of the system, an encapsulated fault tree can be constructed by taking a closer look at the messages that can be sent or received by a capsule. Each received message can be an input failure port and each sent message can be an output failure port of the encapsulated fault tree. The probability of an internal fault and the internal structure of fault trees can be determined by structured techniques like (IF)-FMEA, HiP-HOPS, or HAZOPS (Birolini, 1999; Fenelon & McDermid, 1993; Gomaa, 2000; Liggesmeyer, 2000; Pumfrey, 1999). Based on the encapsulated fault trees, a fault tree for the architecture can be constructed with composition-based techniques, as presented in Grunske (2003c). The resulting fault tree of the level crossing control system is presented in Figure 12. It contains the encapsulated fault trees of the components and connects them with respect to the message flow in the system.

Figure 12. Fault tree of the level crossing control system



Based on this fault tree, the system level failures responsible for accidents can be identified. In the level crossing control system, the safety-relevant failures are (1) to send a faulty green signal to the train when the gates are open or (2) to open the gates when a train is in the level crossing section. The probabilities of these safety critical failures can be determined based on the fault tree and the probabilities of the internal faults as well as the system level input failures given in Table 3. These probabilities are 1,099 $*10^{-5}$ for the first and 4,999 $*10^{-6}$ for the second critical failure. To reduce this failure probability the architecture specification can be restructured. More precisely, the transformation operators Two Channel Redundancy and Multi Channel Redundancy with Voting with two or four ports in the interface graph can be applied to the architecture.

If we assume a probability of 0.5 $*10^{-6}$ for an internal fault in the *voter* or *validation* capsule, the probabilities of the critical failures will be reduced as presented in Table 4. Based on these results, the best architecture transformation is the application of the Multi Channel Redundancy operator to the level crossing control capsule. The resulting architecture after this transformation is presented in Figure 13. The application of the Two Channel Redundancy does not reduce the probabilities of the safety critical failures. The reason for this is the combined probability of an internal fault of the *validation* capsules, which is nearly identical to the probability of an occurrence of a hardware defect in the target capsule.

Table 3. Probabilities of the internal faults and the system level input failures in the level crossing system

Internal faults or system level input failures	Failure or fault probability in 2,000 hours mission time)
TrainSignalControl. HardwareDefect	$1,0 *10^{-6}$
GateControl. HardwareDefect	$1,0 *10^{-6}$
LevelCrossingControl. ConfusionInControlFlow	$0,05 *10^{-6}$
LevelCrossingControl. MessageSentToLate	$0,05 *10^{-6}$
LevelCrossingControl. IgnoredSignalTrainEntered	$0,05 *10^{-6}$
LevelCrossingControl. HardwareDefect	$1,0 *10^{-6}$
GateSensorManager. HardwareDefect	$1,0 *10^{-6}$
TrainSensorManager. HardwareDefect	$1,0 *10^{-6}$
InputFailure.GatesDown	$2,0 *10^{-6}$
InputFailure.GatesUp	$2,0 *10^{-6}$
InputFailure.Entered	$2,0 *10^{-6}$
InputFailure.Left	$2,0 *10^{-6}$
InputFailure.IncomingTrain	$2,0 *10^{-6}$

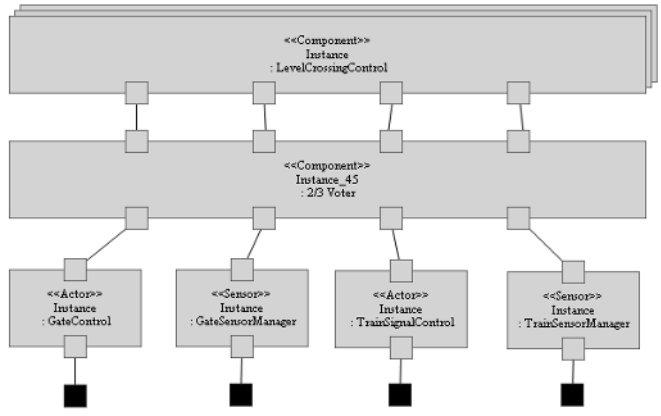
Table 4. Application of the transformation operators to the level crossing control system

Transformation operator (transformed capsule)	Probability of the safety critical failure "A green signal is sent to the train when the gates are open"	Probability of the safety critical failure "The gates are opened when a train is in the level crossing section"
Original Architecture	$\sim 1,099 *10^{-5}$	$\sim 4,999 *10^{-6}$
Multi Channel Redundancy with Voting (TrainSignalControl)	$\sim 1,1 *10^{-5}$	$\sim 4,5 *10^{-6}$
Multi Channel Redundancy with Voting (GateControl)	$\sim 4,6 *10^{-6}$	$\sim 5,0 *10^{-6}$
Multi Channel Redundancy with Voting (LevelCrossingControl)	$\sim 4,5 *10^{-6}$	$\sim 4,5 *10^{-6}$
Multi Channel Redundancy with Voting (GateSensorManager)	$\sim 9,6 *10^{-6}$	$\sim 5,0 *10^{-6}$
Multi Channel Redundancy with Voting (TrainSensorManager)	$\sim 9,6 *10^{-6}$	$\sim 4,5 *10^{-6}$
Two Channel Redundancy (TrainSignalControl)	$\sim 1,1 *10^{-5}$	$\sim 5,0 *10^{-6}$
Two Channel Redundancy (GateControl)	$\sim 1,1 *10^{-5}$	$\sim 5,0 *10^{-6}$
Two Channel Redundancy (LevelCrossingControl)	$\sim 9,6 *10^{-6}$	$\sim 1,1 *10^{-5}$
Two Channel Redundancy (GateSensorManager)	$\sim 1,1 *10^{-5}$	$\sim 5,0 *10^{-6}$
Two Channel Redundancy (TrainSensorManager)	$\sim 1,1 *10^{-5}$	$\sim 5,0 *10^{-6}$

Conclusion and Future Work

In this chapter, the basic concepts of graph-based architecture evolution have been presented. These concepts were extended and adapted to transformation operators improving the quality characteristics of an architectural specification. Consequently, a mapping of the elements of the UML-RT to hypergraph-based elements was presented. Further, the UML-RT was extended to evaluate quality characteristics, such as safety, availability, reliability, and temporal correctness. Based on the mapping of the UML-RT

Figure 13. Architecture specification after the application of the Transformation operator: Multi Channel Redundancy with Voting



to hypergraphs, a set of transformation operators was presented that can be used to improve the quality characteristics of an architectural specification. The practical feasibility of the approach was shown by developing a tool able to propose and to apply transformation operators to an existing architectural specification.

We conclude with some items that remain for future work. First, the transformation operators should be extended by structure-generic graph transformation rules. With these structure-generic graph transformation rules, a component can be identified with the occurrence morphism that can have an arbitrary number of ports. Second, the evaluation models and the set of transformation operators should be extended to evaluate and improve other quality characteristics. Furthermore, the approach and the tool should be applied to other more complex case studies. This will show the scalability of the approach and the performance properties of Balance.

References

Avizienis, A. (1985). The N-Version approach to fault-tolerant software. *IEEE Transactions on Software Engineering, SE-11*(12), 1491-1501.

Birolini, A. (1999). *Reliability engineering: Theory and practice*. New York: Springer.

Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The unified modeling language user guide*. New York: Addison Wesley.

Bosch, J., & Molin, P. (1999). Software architecture design, evaluation and transformation. *IEEE Engineering of Computer Based Systems Symposium*.

Cheng, S. W., & Garlan D. (2001). Mapping architectural concepts to UML-RT. *Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications* (PDPTA 2001), Las Vegas.

Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., & Löwe, M. (1997). Algebraic approaches to graph transformation - Part I: Basic concepts and double pushout approach. In G. Rozenberg (Ed.), *Handbook of graph grammars and computing by graph transformation: Vol. 1: Foundations* (pp. 163-245). Singapore: World Scientific Publisher.

de Alfaro, L., & Henzinger, T.A. (2001). Interface automata. *The Ninth Symposium on Foundations of Software Engineering.* ACM Press.

Douglas, B.P. (1999). *Doing hard time*. Reading, MA: Addison Wesley.

Douglas, B.P. (1999). *Real time design patterns*. Reading, MA: Addison Wesley.

Drewes, F., Hoffmann, B., & Plump, D. (2002). Hierarchical graph transformation. *Journal of Computer and System Sciences, 64*(2), 249-283.

Ehrig, H. (1979). Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, & G. Rozenberg (Eds.), *The First Graph Grammar Workshop* (pp. 1-69). Springer LNCS 73.

Ehrig, H., Boehm, P., Hummert, U., & Löwe, M. (1988). Distributed parallelism of graph transformation. In *LNCS 314, 13th Int. Workshop on Graph Theoretic Concepts in Computer Science* (pp. 1-19). Berlin: Springer.

Fahmy, H., & Holt, R.C. (2000). Using graph rewriting to specify software architectural transformations. *Proceedings of Automated Software Engineering*.

Fenelon, P., & McDermid, J.A. (1993). An integrated toolset for software safety analysis. *Journal of Systems and Software, 21*(3), 279-290.

Gomaa, H. (2000). *Designing concurrent, distributed, and real-time applications with UML*. Reading, MA: Addison-Wesley.

Grunske, L. (2003a). Automated software architecture evolution with hypergraph transformation. *Proceedings of the Seventh International IASTED on Conference Software Engineering and Application (SEA 03)*, (pp. 613-621). Marina del Ray.

Grunske, L. (2003b). Transformational patterns for the improvement of safety properties in architectural specification. In J. Bargary & C. Haskins (Eds.), *Proceedings of the Viking PLOP 03*. Copenhagen: Microsoft Business Press.

Grunske, L. (2003c). Annotation of component specifications with modular analysis models for safety properties. *Proceedings of the 1st International Workshop on Component Engineering Methodology, (WCEM 03)* (pp. 31-41).

Grunske, L., & Neumann, N. (2002). Quality improvement by integrating nonfunctional properties in a software architecture specification. *Proceedings of the Second Workshop on Evaluating and Architecting System Dependability* (pp. 23-33).

Habel, A. (1992). Hyperedge replacement: Grammars and languages. *Lecture notes in computer science: No. 643*. Berlin: Springer.

Harel, D., & Kupferman, O. (2002). On object systems and behavioral inheritance. *IEEE Trans. Software Engineering, 28*(9), 889-903.

Hirsch, D., Inverardi, P., & Montanari, U. (1999). Modeling software architectures and styles with graph grammars and constraint solving. *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, February 22-24 (pp. 127-142).

Hoffmann, B. (2001). Shapely hierarchical graph transformation. *Proceedings of the Symposium on Visual Languages and Formal Methods,* Stresa, Lago Maggiore, Italy, September 5-7 (pp. 30-37). IEEE Press.

Hoffmann, B., & Minas, M. (2001). Transformation of shaped nested graphs and diagrams. In M. van den Brand & R. Verma (Eds.), *Electronic notes in theoretical computer science: Vol. 59*. Firenze, Italy: Elsevier Science Publishers.

Hofmeister, C., Nord, R., & Soni, D. (1999). *Applied software architecture*. Reading, MA: Addison Wesley Longman.

Holt, R. C. (1998). *Structural manipulations of software architecture using Tarski relational algebra.* Working Conference on Reverse Engineering (WCRE '98), Honolulu.

Knight, J.C., & Leveson, N.G. (1986). An experimental evaluation of the assumption of independence in multiversion programming, *IEEE Transactions on Software Engineering*, *12*(1), 96-109.

Le Metayer, D. (1998). Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering, 24*(7), 521-553.

Liggesmeyer, P. (2000). *Qualitätssicherung Softwareintensiver Technischer Systeme*. Heidelberg: Spektrum-Akademischer-Verlag.

Mens, T. (1999). *A formal foundation for object-oriented software evolution.* Unpublished Doctoral Dissertation, Department of Computer Science, Vrije Universiteit Brussel.

Mens, T., Demeyer, S., & Janssens, D. (2002). Formalising behaviour preserving program transformations. In A. Corradini, H. Ehrig, H.-J. Kreowski, & G. Rozenberg (Eds.), *Proceedings of the International Conference on Graph Transformation, Vol. 2505 of Lecture Notes in Computer Science* (pp. 286-301). New York: Springer-Verlag.

Mitra, S., Saxena, N.R., & McCluskey, E. J. (1999). Design diversity for redundant systems. *The 29th International Symposium on Fault-Tolerant Computing (FTCS-29)* (pp. 33-34).

Musa, J.D., Iannino, A., & Okumoto, K. (1987). *Software reliability: Measurement, prediction, application*. New York: McGraw-Hill International Editions.

Papadopoulos, Y., McDermid, J. A., Sasse, R., & Heiner, G. (2001). Analysis and synthesis of the behaviour of complex programmable electronic systems. *Conditions of Failure, Reliability Engineering and System Safety, 71*(3), 229-247.

Pumfrey, D.J. (1999). *The principled design of computer system safety analyses.* Unpublished Doctoral Dissertation, University of York.

Randell, B. (1975). System structure for software fault tolerance. *IEEE Transactions on Software Engineering,* (2), 220-232.

Randell, B., & Xu, J. (1995*).* The evolution of the recovery block concept. In *Software fault tolerance* (pp. 1-21). New York: Wiley.

Rumpe, B., Schoenmakers, M., Radermacher, A., & Schürr, A. (1999). UML + ROOM as a standard ADL? In *Proceedings of the ICECCS '99 Fifth IEEE International Conference on Engineering of Complex Computer Systems,* (pp. 43-54).

Selic, B., & Rumbaugh, J. (1998). Using UML for modeling complex real-time systems. Retrieved July 20, 2003, from Rational Software Corporation, *http://www.rational.com/media/whitepapers/umlrt.pdf*

Selic, B., Gullekson, G., & Ward, P. T. (1994). *Real-time object-oriented modeling*. New York: Wiley.

Taentzer, G. (1999). Adding visual rules to object-oriented modeling techniques. *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS'99)*, Nancy, France. Singapore: IEEE.

Taentzer, G., Ermel, C., & Rudolf, M. (1999). The AGG approach: Language and tool environment. In *Handbook of graph grammars and computing by graph transformation: Vol. 2. Applications, languages and tools*. World Scientific.

Wermelinger, M., & Fiadeiro, J. L. (1999). Algebraic software architecture reconfiguration. In *Software Engineering -ESEC/FSE'99, LNCS 1687* (pp. 393-409). Springer-Verlag.

Wermelinger, M., Lopes, A., & Fiadeiro, J. L. (2001). A graph based architectural (re)configuration language. *Proceedings of the Joint 8th European Software Engineering Conference and 9th Symposium on the Foundations of Software Engineering* (pp. 21-32). ACM Press.

Chapter III

Version Control of Software Models

Marcus Alanen, Åbo Akademi University, Finland

Ivan Porres, Åbo Akademi University, Finland

Abstract

We review the main concepts and algorithms behind a software repository with version control capabilities for UML and other MOF-based models. We discuss why text- and XML-based repositories cannot be used to manage models and present alternative solutions to build a model repository that takes into account specific details of MOF-based modeling languages.

Introduction and Motivation

In this chapter, we study how to store and manage large models during the lifetime of a software project. The first generation of UML editors used to store a whole model as a single file. This approach assumes that once a model is ready there will be no major changes and it can be distributed to the programmers as documentation. Programmers use the model as a reference design or blueprint for the code to be developed, but the model is no longer updated. In this scenario, software evolution and maintenance reverts over to the program source code, not to the UML model.

However, this approach is not satisfactory if we plan to use models instead of source code as the main and most important description of our software. This requires that any model should always be up-to-date. In this context, there will be different developers working

simultaneously on the same models. Different versions of the same model will be targeted to different platforms or customer requirements, and evolution and maintenance will be carried out over the models. This implies that we need to use a proper configuration management system to keep track of our models that comprise the final product.

Configuration management is a well-studied topic, and there are many tools available on the market. It involves several different subtopics such as version control as well as change, build, and release management. Configuration management is a key element in the management of any software development project. It is possible to construct a self-made system using a combination of open-source tools such as CVS, autoconf, make, and Tinderbox, or to use complete commercial solutions such as IBM Rational ClearCase.

However, most of the existing tools are designed to manage either program code or informal documents in natural language. The question now is if we can use existing configuration management systems to keep track of evolving models or if we need new tools and methods customized to the idiosyncrasies of the Object Management Group (OMG) standards. This research is centered on what we consider the central element of a configuration management system for models: a model repository with version control capabilities.

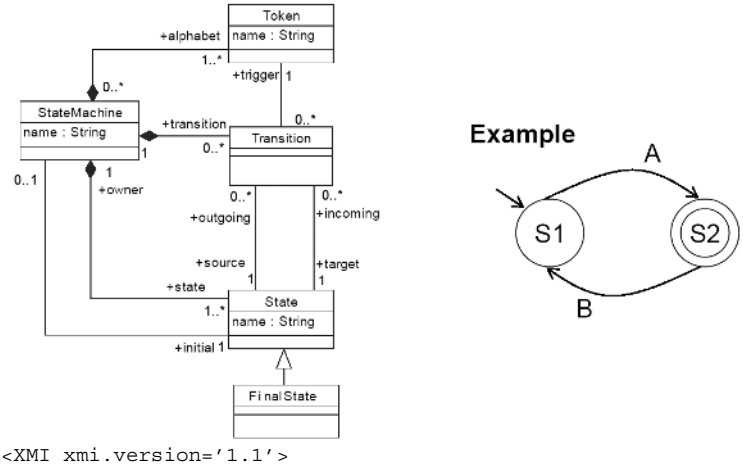
The objective of this chapter is to raise different issues that appear when we try to use inappropriate methods and tools to manage models while discussing possible alternatives that comply with the existing standards.

Modeling Languages and the Meta Object Facility

According to the OMG standards, the information stored in a model is organized internally according to a metamodel. A metamodel describes the abstract syntax of a modeling language. Each class in a metamodel describes a concept or abstraction. Each class may have a number of attributes. An association connecting two classes represents a relation between these elements and is usually split into two opposing properties. A property has several characteristics, such as a name, multiplicity ranges, and directionality. Instances of the elements have connections to each other obeying the properties. The interconnections can be either ordered or unordered, and an element can occur either once or several times.

We can illustrate these concepts in a small modeling language of our invention that is much simpler than UML. Our example language is called FSM and is a language for describing finite state machines. A state machine has a finite number of states and transitions. Each transition connects two states and can be triggered by a token. The set of tokens in a state machine is called the alphabet. One or more states may be marked as final states, while one of the states is marked as initial. These concepts are described as a class model on the left side of Figure 1. We call this kind of diagram a metamodel. This diagram is similar to the metamodels shown in the OMG UML standards. In our example language, the fact that each state machine has an initial state is represented by the

Figure 1. Example model in the FSM language




```
<XMI xmi.version='1.1'>
 <XMI.header>
   <XMI.metamodel xmi.name='FSM' xmi.version='1'/>
  </XMI.header>
<XMI.content>
 <FSM:StateMachine xmi.id='if564' initial='i5044' name='Example'>
  <FSM:StateMachine.alphabet>
  <FSM:Token xmi.id='i8e5d' name='A'></FSM:Token>
  <FSM:Token xmi.id='i606b' name='B'></FSM:Token>
</FSM:StateMachine.alphabet>
 <FSM:StateMachine.state>
 <FSM:State xmi.id='i5044' name='S1' stateMachine='if564'>
```

property named initial. We use the generalization relationship to define a model element as a specialization of other model elements. In our metamodel, a final state is a specialization of state.

The right side of Figure 1 shows an example model in the FSM language. The model is represented using two notations. The diagram at the right uses a syntax that is specific for our language for finite state machines. Most designers would prefer this notation since it is a fully visual language where each concept is described using a different icon, and more importantly, benefits from the human eye's tendency to comprehend diagrammatic data better than text.

However, we can also represent the same model as an XMI document. XMI is an OMG standard (OMG, 2003) for model interchange. It is based on XML and can be used to represent models of any modeling language (i.e., it is not limited to UML). XMI is the preferred notation to exchange models between programs since XMI documents strive to be portable and easy to parse. A full disclosure of the XMI standard is beyond the contents of this chapter; it is sufficient to say that XMI incorporates facilities to exchange abstract models, diagrammatic information, and model differences. An important and oft-forgotten feature of XMI is the concept of XMI.Extension elements, which can be used to add additional metadata when the metamodels used cannot express the desired

qualities. Trivial examples of these extensions include the addition of tool-specific metadata to an element which describes further how it should be implemented or transformed, supplemental graphical information, or the addition of version information to a model. Unfortunately, the current tools lack support for the more advanced features of XMI.

A model or an XMI document describing a model conforms to the metamodel of the model. The UML metamodel is defined in a language called the Meta Object Facility (MOF) (OMG, 2001), which is commonly known as a meta-metamodel. The meta-metamodel gives the power to model the relationships, or, rather, properties, between metamodel elements. For example, the fact that a Statemachine in Figure 1 can own several States is defined by the black diamond, which represents the MOF concept of ownership. Other MOF concepts are the multiplicity ranges, metamodel element names, and the named association ends.

MOF is defined as an OMG standard, and it can be used to define many different modeling languages (i.e., there is nothing specific to UML in MOF). In this chapter, we assume that the models representing our software have metamodels that are described as MOF metamodels. In this sense, this chapter is not specific to UML but to MOF. However, UML is the largest, most used, and best known MOF application, so we will use the UML to illustrate our findings.

A Model-Based Repository with Version Control Capabilities

A configuration management system (CMS) is based on a central repository that contains all artifacts relevant to a software project. We define a model-based configuration management system as a CMS where the project artifacts are structured logically as defined in a metamodel such as the UML modeling language. The central element in a model-based CMS is the repository. The main function of the repository is to store models, but it also has another key function: to enable team work by providing a distributed environment, version control, and change control.

Most repositories are usually distributed systems that allow different developers to work simultaneously on the same project. In this case, the repository resides on (at least) one server and the client computers read and update parts of the repository as needed. We would like that the communication between the repository and the client is based on open standards so we can seamlessly use tools from different vendors. This includes two different standards: a standard to define how and when information is transferred; and a standard to define the format of the information that is being transferred. An example of this distinction is the HTTP and HTML protocols used in Web applications. We can expect that the repository server and the client will use XMI as a common data format. However, there is no standard transfer protocol for XMI models yet. Obviously, a great success of HTTP has been that it does not merely transport HTML, but it transports other kinds of data as well. As such, suitably extended, it could be an appropriate protocol for transporting models as well. The fact that a client and a server use XMI as a model interchange format does not prevent the server to store the models in the repository using

a different format (e.g., to use auxiliary index files to find the information inside an XMI file quickly or even a relational database).

A version control system should store and keep track of different versions of each artifact or document created in a project. A version of a model can represent a different design solution, a different or improved implementation of the same design in a given platform, or perhaps just add more metadata about the design. An important feature of a version control system is the locking scheme. We consider that an optimistic locking scheme, as used in many source code-based versions systems such as CVS or Subversion, increases the level of collaboration and overall productivity of a team and avoids problems such as stale locks. However, an optimistic locking scheme introduces a new requirement into the version control system: a mechanism to compare and merge different simultaneous changes from different developers.

Finally, a change control mechanism defines who can introduce new artifacts and new versions of an existing artifact in the repository and how these changes are reviewed and approved. In larger projects, it may be of interest to restrict the modification of classes or components that are well designed, implemented, and/or tested. In safety-critical systems, the models should not be changed without a formal change procedure.

Commercial modeling tools exist that provide a model repository and version control capabilities in one way or another. It is not in the scope of this chapter to present a review of these tools, but most of these tools present one or more drawbacks that are avoided in this work: they use a pessimistic locking scheme and restrict the version control system to a subset of the UML, usually UML class diagrams. This is precisely the main advantage of our work: it provides a version control system without explicit locks that can be used in any MOF language, for example, all kinds of UML diagrams such as class, state, sequence, activity diagrams, UML profiles, and new languages that have not yet been designed.

The rest of this chapter will discuss some key issues in creating a model-based repository with version control capabilities. In order to implement a repository we need to be able to perform several basic tasks on the elements of a model. First, we must be able to uniquely identify elements in a model, and find elements according to specific criteria. Then, we must be able to further query about the data of the element. Finally, we must be able to change that data. Together, these operations can be used to calculate differences between elements, track element evolution, merge element data and resolve merge conflicts, restrict read and write access to specific parts of a model, transform elements, and enforce a process upon the development of the models.

Element Identification

The most fundamental feature of the repository is to be able to uniquely identify the elements stored in it. One of the most widely used mechanisms to identify an element in a repository is to use a hierarchical naming scheme. The file name C:\My Documents\UML\evolution.tex or the Java class name java.util.Iterator are examples of

hierarchical names. In UML we can create similar names using two colons as a separator. A class named Person inside the package Sales can be referred to as Sales::Person. Hierarchical names are intuitive and easy to use. However, there are two problems with using this mechanism to identify elements in a model.

First, if we rename an element in the model, we lose the linking between the current and the previous version. As an example, if we rename the class Person to Customer, there will be nothing in our repository that would tell us that Sales::Customer is actually derived from Sales::Person. The solution would be to somehow add this missing information to the repository. However, there is yet another problem: not all the UML model elements have proper names. For example, generalization relationships are never named. The same applies to transitions in a statechart, links in a sequence diagram, and many other minor but equally important elements. Consequently, the identification of elements cannot depend on the metamodel but must work as a generic solution for any (MOF) metamodel.

The solution is provided by the XMI standard itself. The standard states that each element in a model may have a Universally Unique Identifier (UUID) (pp. 1-3 of (OMG, 2003)). An example UUID is the string DCE:2fac1234-31f8-11b4-a222-08002b34c003. UUIDs are assigned the first time that an element is exported to an XMI document. Later, any standard-compliant open tool that imports the XMI document should not change or remove the assigned UUIDs. Primitive types, such as integers or strings, are special and cannot be treated in the same manner as model elements. Thus, they do not acquire unique identifiers.

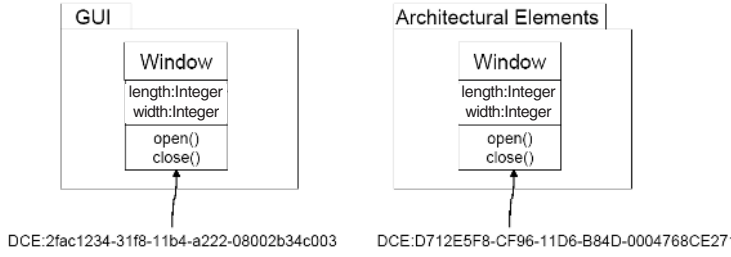
UUID strings in the DCE namespace (CAE Specification, 1997) are assumed to be globally unique, even across models, tools, and time. They are based on a 128-bit pseudorandom number generated from the physical address of the network interface in the host running the tool and the tenths of microseconds elapsed since the Gregorian reform (15 Oct. 1582). The UUID strings are long and too complex to be generated by hand, but this is not an issue for the user since the UML tools should take care of this aspect. Unfortunately, many of the existing UML tools do not generate or even preserve UUID strings. To check this, we can perform a simple test. Use your favorite CASE tool to generate a small model (it can be empty) and save the model in an XMI file. Edit the XMI file with a text editor and add a UUID identifier to the Model element. For example:

```
<UML:Model xmi.id = '122' name = 'Example Model' isAbstract = 'false'
  xmi.uuid = 'DCE:2fac1234-31f8-11b4-a222-08002b34c003'
   isRoot = 'false' isLeaf = 'false' isSpecification = 'false'>
```

Then save and close the file in your text editor and load it in the CASE tool. The modified XMI file should be imported without problems. Export the model again to the XMI format and open the XMI file with your text editor. Examine the line defining the Model element. The UUID should be there, unaltered. If the CASE tool has modified the UUID string or has removed it completely, then it does not comply with the XMI standard for open tools.

UUID strings allow us to differentiate between two instances of the same element and two elements that are similar. We consider that two model elements of the same type and

Figure 2. Unique identification of elements using UUID



with similar properties are still two different model elements if their UUID strings are different. An example of this is illustrated in Figure 2. This contains two classes that have the same name and properties but that represent two different abstractions from two different problem domains.

This distinction is fundamental to implement model management operations like comparing two models, merging two models into one, or duplicating/deleting (parts of) a model. If we merge the two models represented in Figure 2 into one, we want to keep two different classes named Window, since they represent two different things.

In other cases, model elements with the same name do actually represent the same abstraction. One typical example is the standard classes predefined in a programming language such as Integer or java.util.Iterator. The two models from Figure 2 implicitly contain two classes named Integer, which probably are the same concept. If we merge the two models, the result should contain only one class named Integer. We can solve this problem by assigning a predefined identifier to standard elements such as the libraries of programming languages. In fact, the Java Object Serialization Specification states that each (serializable) Java class has a unique 64-bit integer used to uniquely identify the class in a stream. We could derive the 128-bit UUID from the 64-bit Java identifier. However, the XMI standard does not describe how to do this. Also, other programming languages like C++ do not have assigned identifiers for their library elements. A possibility would be to identify such elements by name, (e.g., c++.std.iostream.cout), but this is exactly what we are trying to avoid by using UUID strings! The solution may be to standardize a method to create UUID strings once based on the name and signature of these classes (e.g., using MD5 hash sums). This way it could be possible to generate automatically UUID strings for the standard library of languages such as C++.

The next question is what happens when we have two instances of a model element, with the same UUID but different properties. Since we assume that UUIDs are unique, we have two versions of the same element. In this case, we want to be able to detect that the element has been changed and to calculate what actually has been changed.

Version Control

A version control mechanism keeps track of what has changed in different versions of a configuration item. It can also combine different changes into a new item. In the context of a model-based repository with optimistic locking, version control is provided by the possibility to calculate the difference between several versions of a model and to combine a difference between models into another model. As an example, assume that the original model shown at the top of Figure 3 is edited simultaneously by two developers. One developer has focused his work on the classes A and B and decided that the subclass B is no longer necessary in the model. Simultaneously, the other developer has decided that class C should have a subclass D. The problem is to combine the work of both developers into a single model. This is the model shown at the bottom of Figure 3.

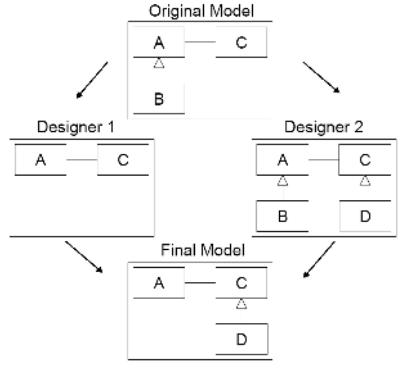
Traditional version control systems work with larger program units such as the file level. In the case of UML, this would be equivalent to performing version control at the diagram level. However, there is a need for fine-grained version control. Coven (Chu-Carroll & Sprenkle, 2000) also implements what its authors call fragment-based versioning. According to them, coarse version control systems produce change sets that are too large and difficult to integrate.

Difference Between Models

Computing the differences between two models or two versions of the same model is a fundamental operation in a version control system. This basic operation allows us to define the evolution of a model as the sequence of differences between two consecutive versions of the model.

The implementation of this operation may seem trivial. Since XMI files are basically text files, we could try to use a tool designed to compute differences for line-based text files such as source code to analyze our models. The UNIX programs diff and patch are two

Figure 3. Union of multiple versions of a model



fine examples of these tools. The problem is that line-based tools will detect false changes in a model. For example, file1.xml and file2.xml as shown in Figure 4 represent the same UML model, probably generated by two different UML editors. However, a line-based tool such as diff considers these files different.

The next possibility is to use an XML-based tool. Such a tool should be aware that the previous example represents the same document. Still, an XML-based tool is not aware of special features of a metamodel such as whether or not some elements in a model are ordered. For example, the order in which the classes in a package are defined is not relevant in a UML model. Considering this, file3.xml and file4.xml as shown in Figure 5 represent the same UML model, although they are two different documents at the XML level. In other cases, such as the definition of the parameters in a method of a class, the actual order of the parameters is relevant. So, using an XML-based tool that simply ignores the order in which elements are defined is not a solution to this problem either. The ordering information of metamodel associations is only available in the metamodel and will be ignored by a generic XML tool.

As a consequence of the previous discussion, we can only compute differences between two models by using algorithms specifically designed to handle XMI models and only when these algorithms use information about the metamodel of a given modeling language.

An algorithm for calculating the difference between two arbitrary models is described in Alanen and Porres (2003). In this chapter, we present an application of these algorithms

Figure 4. The same model in XMI as two different ASCII files

```
file1.xml:
  <UML:Model xmi.id = '122' name = 'Example Model'/>

file2.xml:
  <UML:Model name = 'Example Model' xmi.id = '122'/>
```

Figure 5. The same model in XMI as two different XML documents

```
file3.xml:

<UML:Model xmi.id = '122' name = 'Example Model'>
  <UML:Namespace.ownedElement>
    <UML:Class xmi.id = '123' name = 'Customer'>
    <UML:Class xmi.id = '124' name = 'Product'>
    ...

file4.xml:

<UML:Model xmi.id = '122' name = 'Example Model'>
  <UML:Namespace.ownedElement>
     <UML:Class xmi.id = '124' name = 'Product'>
     <UML:Class xmi.id = '123' name = 'Customer'>
     ...
```

in a model-based repository. We will denote Δ to describe a difference between two models. The example described earlier in Figure 3 can be decomposed into three tasks. Calculate the difference Δ_1 between the model from Designer 1 and the original model (see the bottom of Figure 6), calculate the difference Δ_2 between the model from Designer 2 and the original model (see the top of Figure 6) and finally, merge the original model with the two differences (see Figure 7). The result of a difference is not always a model, in a similar way that the difference between two natural numbers is not always a natural number but can be a negative one. An example of this is shown in the bottom part of Figure 6. In this case, the difference of the models contains negative model elements (i.e., elements that should be removed from a model).

In a version control system we require two basic algorithms for model difference and model merge; given any two models M_{old} and M_{new} and a difference Δ between them, they have the following properties:

$$M_{new} - M_{old} = \Delta$$
$$\Delta(M_{old}) = M_{old} + \Delta = M_{new}$$

The Δ is a transformation that when applied to model M_{old}, yields model M_{new}. The internal representation of Δ and the operators + and - can be defined in several ways. Figures 6 and 7 suggest that the granularity of the components for a difference would be parts of models, and indeed that is the approach taken by the XMI standard. However, using even more basic building blocks has its advantages, as we will see next.

Figure 6. Difference of models

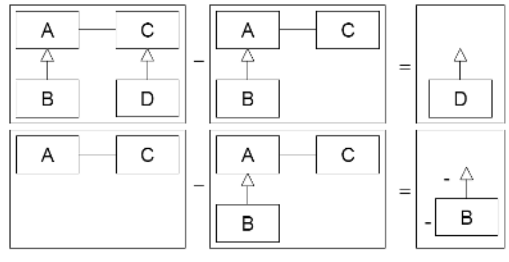
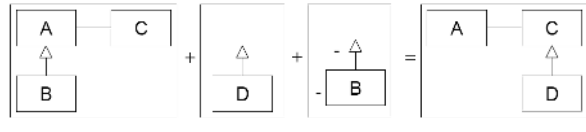


Figure 7. Union based on differences



Representing and Storing Differences

We have shown informally that the result of a difference between two models may contain negative elements (i.e., information that should be removed from a model). We further realize that it is very common to only have tiny parts of the model changed, especially merely interconnections between elements. Thereby we consider that it is intuitive to represent a Δ in operational terms; not as a set of elements and negative elements but as a sequence of transformations that add or remove elements from a model.

We have identified seven elementary transformations in a model that will be used as the basis for defining a Δ. We assume that it is not possible to change the type of a model element (e.g., a UML Class cannot become a Package, and an element cannot change its UUID). Yet an important assumption is the concept of "zero" values, denoting a well-known value independent of the metamodel or model but dependent on the datatype (an empty string, list, set, or the number zero). The operations are split into two categories; the first category delves into the lifetime of elements and consists of two operations:

- **create(e,t) :** Create a new element of type t with the UUID of e. By default, a new element has all its features set to their default values.
- **del(e,t) :** Delete an element of type t with the UUID of e. An element may only be deleted if all its features are set to their default values.

The second category consists of operations which modify slots, in practice the interconnections between elements. Here, modifications of a slot of property type p of an element e with UUID u are done. Where necessary, there is another element e_t with UUID u_t. Depending on the type of the property, this might mean one of the following modifications:

- **set(e, f , v_o, v_n) :** Set the value of e.f from v_o to v_n, for an attribute of primitive type.
- **insert(e, f , e_t) :** Add a link from e.f to e_t, for an unordered property.
- **remove(e, f , e_t) :** Remove a link from e.f to e_t, for an unordered property.
- **insertAt(e, f , e_t , i) :** Add a link from e.f to e_t, at index i, for an ordered property.
- **removeAt(e, f , e_t , i) :** Remove a link from e.f to e_t, which is at index i, for an ordered property.

It should be noted that none of the operations try to maintain the bidirectionality of relations. It is maintained at a higher level in the actual difference algorithm. Also, for transferring operations over the network, the UUID of an element must be used instead of the actual element.

These operations have two important properties. First, the positive operations (create, set, insert, and insertAt) are complete in the sense that they can be used to represent any

model. Also, each operation has a dual operation with the opposite effect. The map between operations and their dual operations is given in Table 1. This is needed to calculate the inverse of a Δ. An inverse effectively cancels a previously applied Δ and is useful for empowering developers to back out bad changes.

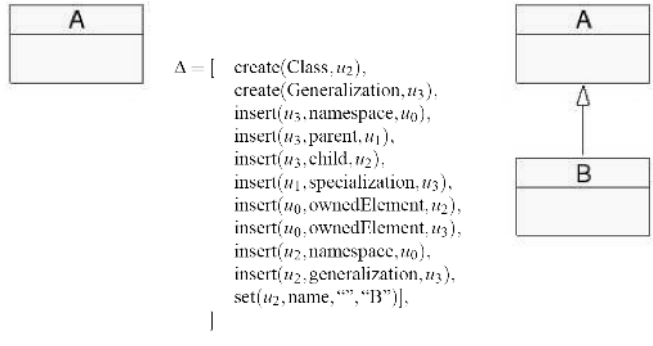
An example of a difference between two models is given in Figure 8, in which the old model is on the left and the new model is on the right. In the Δ, two new elements u_2 and u_3 are created. They are connected to the root Model element u_0 (not shown) via their namespace association, and the Model connects them to its ownedElement composition, due to bidirectionality constraints. The new class u_2 is connected to the old class u_1 via the Generalization element, using the specialization/parent and generalization/child properties. Finally, the name of the new class is set.

The XMI standard describes a system to represent differences in a model inside an XMI document (pp. 1-32 of (OMG, 2003)). According to the standard, a difference entity can be used to add, delete or replace an element in a model. Although this approach is valid and can be used to describe differences, it is very coarse-grained. If we just want to represent that a class has changed its name, we need to replace the complete class in the model. We consider that the various elements of XMI.Difference should be specialized into basic operations that work at the property level instead of the element level. An example of how the difference in Figure 8 could look in this XMI is given in Figure 9. (Please note, we have shortened the UUID strings for clarity.)

Table 1. Map between operations and dual operations

Operation O	Dual operation Õ
create(e,t)	del(e,t)
del(e,t)	create(e,t)
set(e, f , v_o, v_n)	set(e, f , v_n, v_o)
insert(e, f , e_t)	remove(e, f , e_t)
remove(e, f , e_t)	insert(e, f , e_t)
insertAt(e, f , e_t , i)	removeAt(e, f , e_t , i)
removeAt(e, f , e_t , i)	insertAt(e, f , e_t , i)

Figure 8. Difference between two simple models



By using finely grained differences we reduce the traffic between the repository and the clients as well as increase its overall performance, since we have actually reduced the total amount of difference information. In a distributed setting, network communication will be substantially faster as model differences are often significantly smaller than complete models. Also, for visualization purposes, it is important to keep the difference as small as possible in order to aid the developers.

Merging

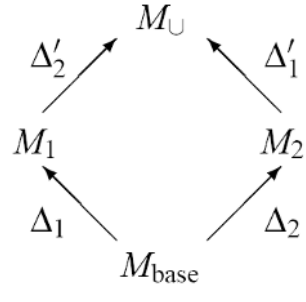
Conventional repositories often enable users to use optimistic locking. This means that several developers can check out and edit the same file, and the changes are then interleaved together into a final merged version. In case the same parts of files were modified and the conflict resolution algorithm of the repository cannot correct the problem, a conflict has occurred and it is the task of one of the developers to rectify the situation before proceeding further.

The same applies to modeling, as could be seen in Figure 3. A merging facility is necessary, as well as systems for conflict detection, automatic resolution, and manual correction. One possibility of a merging facility was introduced in Alanen and Porres

Figure 9. Example of the Δ from Figure 8 described in a modified XMI syntax

```
<XMI.Difference>
   <XMI.create type = 'Class' uuid = 'u2' />
   <XMI.create type = 'Generalization' uuid = 'u3' />
   <XMI.insert from = 'u3' name = 'namespace' to = 'u0' />
   <XMI.insert from = 'u3' name = 'parent' to = 'u1' />
   <XMI.insert from = 'u3' name = 'child' to = 'u2' />
   <XMI.insert from = 'u1' name = 'specialization' to = 'u3' />
   <XMI.insert from = 'u0' name = 'ownedElement' to = 'u2' />
   <XMI.insert from = 'u0' name = 'ownedElement' to = 'u3' />
   <XMI.insert from = 'u2' name = 'namespace' to = 'u0' />
   <XMI.insert from = 'u2' name = 'generalization' to = 'u3' />
   <XMI.set from = 'u2' name = 'name' oldvalue = '' newvalue = 'B' />
</XMI.Difference>
```

Figure 10. The principle of calculating the union of two models, given their base model. Either difference is modified according to the other one, and then applied.



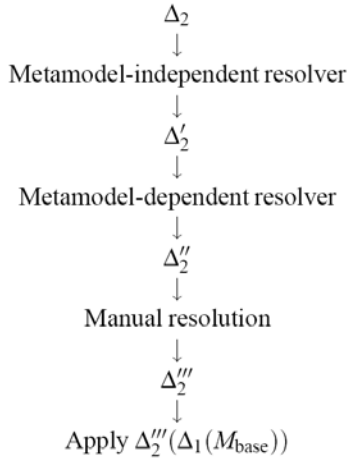
(2003). In the context of a base model and two different changes, Δ_1 and Δ_2, the main idea can be illustrated in Figure 10; either of the Δ_i is modified according to the other, and then applied after the other has been applied. Naturally, when the Δ_i do not overlap, no modifications are necessary, so the nontrivial cases are when the changes overlap for various properties. In the paper, several shortcuts were presented in a metamodel-independent way to allow some automatic conflict resolution. However, far from everything can be automated, and unfortunately this leads to conflict situations akin to those found in ordinary line-based repositories which the developer must correct.

There are several cases where merge conflicts are a fact and manual resolution is required. Modifying the same attribute or the same ordered slot easily creates such situations. For association slots, the opposite slot also must be kept in synchronization. The extreme case of deleting an element even though another Δ merely modifies it slightly leads to a complex question: Which change should be prioritized? Further work in this area is clearly required as automatic conflict resolution is an important feature in any collaboration platform. Again, a pure XML-based approach to conflict resolution is not as thorough as one with knowledge of the metamodel. This is due to the fact that XML considers all properties to be ordered, even though some are not. A great number of seemingly conflicting cases can be resolved automatically if the property under modification is actually unordered.

We have developed a conflict resolution mechanism that has three distinct steps 1) A metamodel-independent resolution step, 2) a metamodel-dependent step where the conflict resolution algorithm takes the metamodel of the elements and their well-formed rules into consideration, thus providing automatic resolution where possible, and 3) a step for manual resolution by the developer. Naturally, the work to be done should become smaller in each step for this to be a viable mechanism.

The second step in the conflict resolution mechanism can also include specific heuristics for conflicts depending on the metamodel. A prime example is diagrammatic information,

Figure 11. A complete merge system with three distinct resolution steps



as the diagram elements themselves do not have any semantic meaning, so the features of the diagram elements are not nearly as correctness-critical as the underlying model. For example, conflicting diagram element coordinates on the diagram canvas can more or less be completely ignored by modifying Δ_2 suitably. Clearly, there is a strong need for metamodel-specific resolvers.

The schema in Figure 11 summarizes a merge system for models. The difference under modification, Δ_2, passes through several filters which modify it to better fit $\Delta_1(M_{base})$. Obviously, all possible mechanic resolution mechanisms should be tried before manual resolution is used.

This simple repository is too coarse-grained for most practical uses. It also lacks many important features. We may want to use the repository to keep a history of the evolution of a model through the whole development cycle. In this case, it is important that we are able to identify, version and retrieve each individual element in a model.

A Model Repository

The task of a model repository is to store successive versions of a model and retain old versions. A simple model repository can store each version of a model as a different file containing the model as an XMI document. In such a system, the file name can be used to identify each version of a model in the CMS. Access control to the repository is managed by the access control mechanism of the filesystem.

Model Storage

While a filesystem still can be used for all this, it is not efficient and problems can occur with respect to atomicity and concurrency. An alternative is to store models into a database. A database can set arbitrary rules for access, modification, and retrieval; transactional properties such as atomicity, consistency, isolation, and durability are guaranteed by the database backend. Also, it is easy to store additional metadata of the models.

Relational databases are not specially well suited to store hierarchically structured information, which models are. The upside is that relational databases have been researched very thoroughly and industry has greatly invested in creating highly scalable and efficient products. The downside might be that model information is inherently object-oriented and does not map naturally into the relational model.

The advent of XML has spurred research in databases particularly suited for storing large XML documents (Jagadish et al., 2002). XML repositories are very similar to object-oriented databases (OODBs), and share their benefits and ills. Among the benefits are much more flexible arrangements of data, ways to manipulate that data and more complicated queries. However, current technology does not scale as well as relational databases. Query optimization, especially, is not as well-known as in the relational

database field. Using an XML database itself could be a great advantage, but until technology catches up, it cannot be deployed for large-scale projects.

In the next subsection, we take a closer look at the relational database schema required for a model repository. It serves merely as an example skeleton of how to map hierarchical information to relational space.

Relational Database Schema

Formally, a relational database consists of relations, tuples, and attributes. Each relation is defined by its name and its named attributes. A tuple is a record of the database (i.e., the actual data we are storing). The data can be cross-linked between several relations. Retrieval consists of fetching the tuples matching a certain query, often speeded up by matching attributes which are primary keys (i.e., unique values in the relation). The problem of storing models in a relational database is fundamentally about mapping a graph with different kinds of nodes and edges to a relational model.

The relational database schema consists of a static set of relations independent of the metamodels used, and a set of relations for each element in every metamodel used (Alanen, 2002). The static set consists of database tables that maintain the version history of the models and enables arbitrary elements to connect to other elements, whereas the rest consists purely as containers of primitive model data such as strings, integers or enumeration values.

There are two strategies to store the different versions of a model element. The first is to store each individual version as a different element including all its attributes. The second alternative is based on the previous discussion on differences between models. We can store only the difference between two versions of a model instead of the complete model elements.

If we store complete model elements, the database will require more space. If we store only the differences between revisions of an element, the size of the database will be smaller, but the queries will be more complex and require more processor time. To simplify our exposition, we present only a database schema where model elements are stored completely. We also do not discuss tables required for a full repository implementation requiring transaction histories, branching and access control, and omit various string concatenation encoding rules.

In the database schema, the table Model consists of a map between model revision to element revisions. The ElementType table contains a row for each model element in a model. It has two columns, the UUID of the model element and the type of the model element. We assume that an element cannot change its type. Each revision of an element receives a revision number from the version database sequence which is unique for the repository. To know what UUID a revision has, we must keep a RevisionUuid table which maps revisions to UUIDs.

A revision of an element consists of data of primitive type, and of links to other elements. These are collected into two different tables, of which one is specific to the metamodel type, and the other is generic. The name of the specific table is created based on the full

name of the modeling language and the name of the metaclass. For example, the table Elements_UML15_SimpleState is used to store SimpleState elements from the UML 1.5 language and Elements is just a prefix to avoid name collisions with other tables unrelated to the UML 1.5 language. Each row in this table will represent a specific revision of a model element of that given metaclass. The primary key of the table is the revision number of the model element. The other attributes are the properties of primitive type of the element. Another example of a metamodel-specific table can be seen in Figure 12.

The generic table is called Connections and maps the connections between elements. Since a slot of an element contains a set or sequence of references to elements, it is important to be able to remember the position of elements for ordered connections. Thus, Connections is a relation (revision, name, target, index). Here, revision is the repository-unique version identifier of our element. Name represents the connection name, for example, ownedElement for a Package owning several classes. Curiously, target must reference the UUID of the target element, not its repository version identifier. This is inherent in the bidirectionality of connections in most modeling languages. If we had two interconnected elements A and B and repository revisions were used exclusively, and A were to change, its revision identifier changes, and thus the database tuples of B would change, resulting in a change of its revision identifier as well! This would cascade through the whole model and create new revisions of every element in the model. This is clearly not desired, and thus links must be from a revision to a UUID. Then RevisionUuid and Model can be used to retrieve the actual revision to use. The final column, index,

Figure 12. An extract from an Elements_UML15_Class table. Any attributes of primitive type are present in the table. The rows are indexed by the element revision, which is a primary key in this table. The enumerations index into their respective types table (e.g., Enum_UML15_VisibilityKind).

revision	name	visibility	isAbstract	. . .
5	Class1	public	false	. . .
9	Class2	protected	true	. . .
13	Class3	private	true	. . .

Figure 13. An extract from a Connections table. Parameters with UUIDs E_2, E_4, E_3 (in that order) of revisions 2, 4, 3 are bidirectionally connected to E_1 with revision 1. Notice the usage of the index attribute to maintain the correct order.

revision	name	target	index
1	parameter	E_2	0
1	parameter	E_3	2
1	parameter	E_4	1
2	behavioralFeature	E_1	-1
3	behavioralFeature	E_1	-1
4	behavioralFeature	E_1	-1

simply keeps track of which element should be in which position in the connection. This is required since few (if any) relational databases keep their records in order. An example of a Connections table can be seen in Figure 13.

Naturally, there are several more ways to encode the same information, and several optimizations that could be made, especially space-wise. For example, since the set of possible connections are known, we can also separate Connections tables for each connection, in this case, Connections_UML15_Package_ownedElement, thus making some of the information implicit. Another trivial optimization is to split the Connections table in two, one for all unordered connections (thereby making the fourth attribute index unnecessary), and one for all ordered connections. The main drawback of these optimizations is that the amount of database commands that must be used increases, and thus there is a risk for an effective slowdown of the repository. This, however, can only be determined by empirical tests since the actual performance varies from one backend to another.

In order to aid in debugging the database, several string constants are kept in some tables, and rows of the other tables reference these strings. Each enumeration is kept in a separate table, e.g., Enum_UML15_VisibilityKind, and the enumeration strings for attributes in the element tables keep references to those strings. Thus, the user debugging sees the actual enumeration strings instead of obscure enumeration numbers, while still keeping the memory requirements low. All element names and all connection names are also kept in separate tables, ConnectionNames and MetaclassNames, for the same reason. This might even save database memory and make the processing faster.

To clarify, the tables discussed in this section have been collected. Where possible, the database has received hints of which columns can be hashed, made primary keys, or reference other tables. Hashes group together similarly hashed columns, primary keys create unique rows in a table, and references provide integrity between the various database tables. These common database layout enhancements speed up queries considerably, and provide some referential integrity in the database. They are marked with the keywords hashed, primary and references table name.

- Model = (model_revision **hashed**, element_revision)
- ElementType = (UUID **primary**, type **references** MetaclassNames)
- RevisionUuid = (revision primary, UUID)
- Connections = (revision **hashed**, name **references** ConnectionNames, target, index)
- ConnectionNames = (name **primary**)
- MetaclassNames = (name **primary**)

Additionally, one table for each metamodel class or enumeration is required.

Access Control

Access control defines the mechanisms by which read or write access to parts of the model are defined and modified. Access control can be implemented at different levels of granularity. The access control level that is easiest to understand and implement is access control at the model level. In such a mechanism the repository may grant read-only or read and write rights to a whole model for a specific set of developers.

However, in some projects, limiting access for developers to only some parts of a model may be important, or even mandatory. As an example of limiting read access, security-related information is to be disclosed only to a specific set of developers. A more common scenario is limiting write access, such that a group of developers may work on a part of a model, and another group on another part. In such cases, it might feel intuitive to set the granularity of access at the element level, whereby read or write access is determined based on the elements that a developer wants to read or change. However, this may be impossible due to fact that associations in the metamodel are relations. Each metamodel association is represented as one property in each participating class. Modifying an association implies the modification of the two associated properties.

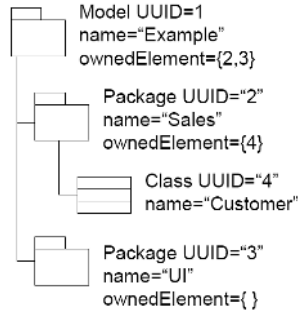
For example, consider a class which has write restrictions. In UML, this class is represented as different properties, including its name, attributes, superclasses, and subclasses. It is then impossible to create a new class as a subclass of this write-restricted class, since that requires modification of the specialization property of that class – which the developer is not allowed to modify! However, most developers would consider these changes as harmless to the original class. This is because while some properties carry semantic meaning for an element, other properties only act as a navigational aid or as the opposite end of a bidirectional meta-association.

Clearly, the level of detail in access control must be based on the properties of elements, not on the elements themselves. In some cases, the developer ought to be able to use a class, subclass it but not add new operations or change existing attributes. The distinction cannot be made by allowing or disallowing write access to the class element itself, but the properties of the class.

Client-Server Communication

As already mentioned, XMI as such does not define a protocol for transferring models over a network, only the encoding of a model. In the interest of software compatibility common standards ought to be defined. Special interest groups, separate from OMG, are advancing the state of the art of distributed authoring, and are creating official Internet standards to fill this void. Good examples are the IETF WebDAV and Delta-V working groups, which have defined "HTTP Extensions for Distributed Authoring - WEBDAV" (RFC 2518) (Goland, Whitehead, Faizi, Carter, & Jensen, 1999) and "Versioning Extensions to WebDAV" (RFC 3253) (Clemm, Amsden, Ellison, Kaler, & Whitehead, 2002) to ease communication in a distributed development environment. These standards can be used for a protocol between a model repository and the client tools, such as a UML editor.

Figure 14. Example repository client access



It is beneficial to examine in more detail what operations especially related to models these relatively new protocols should provide in order to become de facto standards of model transportation.

Most of the time, a client will not be interested in all the elements in a model but only in a subset of them. The problem is that a client might not know the name or the UUID of a certain model element in which it is interested. The reason the client cannot download the whole model is due to the comparatively low bandwidth between the client and the server. There are two main solutions to this problem: One is to let a client seek elements in the model; the other is to implement a query language.

In the first solution, the server should provide a simple interface with two services: one service, named getRoot, returns the UUID of the root element in a model, while the second service, getElement, accepts a UUID as a parameter and returns the model element associated to it. As an example, we can assume our repository contains a simple UML model with two packages and one class as shown in Figure 14. For simplicity, we use integers to denote the UUIDs of the model elements.

In the example repository, if a client invokes the service getRoot the repository will return the value '1,' the UUID of the main element in the repository. If the client invokes getElement('1'), the repository will return all the properties of the element with UUID '1,' including the property named ownedElement. In UML, this property describes the contents of a model or a package. In the repository the model contains the packages Sales and UI, therefore the ownedElement property will be the set {'2,' '3'}, the UUIDs of the previous packages. The client can use these UUIDs to continue traversing the models. This interface can be naturally extended to include revision numbers, for example, getRoot(5) will return the root element in the 5th revision of the model.

An alternative solution is to use a query language, something akin to the SQL in the world of relational databases. In this case, the client will send a query as a text string to the repository that will evaluate the query against all elements in the model and return those who satisfy it. We can use different alternatives as a query language. OCL (Warmer & Kleppe, 1998) is used in the UML metamodel to define additional constraints over valid UML model elements, but also can be used as a query language. As an example, if a client

sends the query self.oclIsKindOf(Class) and self.name="Customer" to our example repository, the repository will return the set {4}. The result is a set since there can be more than one class with the name Customer.

Unfortunately, most UML practitioners are not familiar with OCL. Also, the current OCL parsers are not as optimized as the existing database engines. This is due to the fact that we still do not know which are the most common queries that should be optimized. Finally, we would need to extend the current OCL standard with queries to retrieve version information so we can perform queries against the version history of the repository such as

self.name = "Customer" and self.lastEdited < "1 Mar 2004"

An alternative to OCL is to use an XML-based query language such as XQuery (W3C, 2003). However, the syntax of XQuery and other XML-based languages is too cumbersome. Currently, parsing and compiling technology is so advanced and desktop computers so powerful that there is no reason to obfuscate the syntax of a computer language to make it easy to parse by a computer. Also, this approach does not solve the need to know how the model information is arranged in the UML metamodel in order to create a complex query.

Model Evolution

The basic algorithms for model difference and merge allows us to know that a model has changed, however, quite often we need to know *why* it has changed. For this, additional metadata is required. While an informal description of the change goes a long way, formal, traceable reasons for the change are a boon to bring software engineering toward a robust scientific discipline. Also, we would like to keep the history of the model, and review several old versions of it.

Quite often, a new version of an element represents just an improvement from the previous version. But in many other cases, a new version of an element is derived from one or more other elements, possibly of a different type. This is the case, for example, when we create a new class that realizes the functionality described in a use case or create a statechart as a refinement of another statechart. A version control system keeps track of edited elements but not of derived elements, nor the reason why they have been created. Previous versions of the UML provided a model element named Flow to model evolution relationships. However, it seems that this element has been removed from UML 2.0. It was not supported by the main UML tools, and in any case it, was not useful to create traces between elements that resided in two different models or in models that were described in different modeling languages.

The long-term solution seems to be in yet another standard. The OMG has a request for proposals for a query, view, and transformation language for MOF 2.0 (QVT) (OMG, 2002). One of the operational requirements for the proposals is the ability to trace the execution of transformations. This can be achieved by defining a tuple (S, T, r), where

S and T represents sets of model elements and r a transformation rule such that r(S) = T. This can also be generalized to allow free-form editing of a model as a possible transformation.

Another requirement for the proposals is that they should provide a MOF-based metamodel for the proposed language. The actual metamodel varies from one proposal to another, but the important implication is that the MOF standard describes how to generate XMI documents from any MOF model. Once the standard is accepted, we will be able to represent the evolution history of a model as a sequence of transformation traces, and we will be able to store the evolution history as a standard XMI document.

Conclusion

A software repository is an essential element in any software development project where there is more than one person involved. If we follow a model-based development approach such as MDA (OMG Architecture Board, 2001), we may require a software repository that can manage and version software models since models are the primary artifact representing our software. Also, in an MDA project, models are created and updated constantly, including multiple versions of the same model that are created simultaneously in order to separate the problem space from the implementation concerns.

The construction of a repository that supports model-based development and model evolution may not seem an interesting problem. Many similar systems to manage source code and XML documents already exist. XMI, the standard model interchange format, is based on XML so it may seem that we can simply use any of the existing XML systems to manage our models. However, we have seen in this chapter that XML tools may be too generic and that there are open issues not addressed by the standards. The solution is to implement a repository and version control system for models that is aware of the features of the modeling language used to create the models. That is, to create a repository that manages models using information about the metaclasses and meta-associations of a given metamodel.

Once we assume that XML-based tools do not provide a good foundation for managing our models, we can envision new features for a model repository that are not present in current systems. Specifically, we have discussed the basic algorithms to perform version control of models with optimistic locking. This is an important feature that has become de facto in most source code-based repositories, but it is not trivial to implement in a metamodel-independent way. A model-based repository may also provide additional features such as a search for specific model elements in a large repository using a specialized query language or partial transfer of models between the repository and the clients.

We have also discussed that while a model repository and its clients can use XMI as a model interchange format, there is still the need to identify the transfer protocol. Probably, there is no need to create a new protocol since we can use existing protocols such as WebDAV for this task. However, the OMG standards should define which protocol should be used, independently if this is a new or an old protocol, in order to

guarantee interoperability between clients and repositories from different vendors. Also, there is no special advantage for a repository to store models internally using XMI. We have presented an alternative storage model based on a relational database that may be more efficient. However, the actual performance of the repository depends on many components, including the available network bandwidth and the database backend. Also, we expect that in the future XML-based databases (Jagadish et al., 2002) may outperform relational databases for this specific task.

The open question is what the responsibility of the repository for maintaining the models in a consistent state is, especially maintaining consistency between structural and behavioral diagrams and between diagrams and program code. One possibility is to follow the same approach as in source code repositories: It is the responsibility of the clients to ensure that the models stored in the repository are consistent. However, the other alternative opens an interesting research problem and future direction for our work: How a repository can determine whether a model is consistent or not and which model transformation and code generation steps can a repository invoke in order to make a model consistent.

References

Alanen, M. (2002). *A meta object facility-based model repository with version capabilities, optimisticlocking and conflict resolution.* Unpublished Master's Thesis, Åbo Akademi University.

Alanen, M., & Porres, I. (2003). Difference and union of models. *Proceedings of the UML 2003 Conference,* October.

CAE Specification. (1997). *DCE 1.1: Remote procedure call.* (1997). Available at *http://www.opengroup.org/onlinepubs/9629399/toc.htm*

Chu-Carroll, M. C., & Sprenkle, S. (2000). Coven: Brewing better collaboration through soft-ware configuration management. *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: 21st Century Applications,* November.

Clemm, G., Amsden, J., Ellison, T., Kaler, C., & Whitehead, J. (2002, March). Versioning Extensions to WebDAV, RFC 3253. Available at *http://www.ietf.org/rfc/rfc3253.txt*

Goland, Y., Whitehead, E., Faizi, A., Carter, S., & Jensen, D. (1999, February). HTTP Extensions for Distributed Authoring — WEBDAV, RFC 2518. Available at *http://www.ietf.org/rfc/rfc2518. txt*

Jagadish, H., Al-Khalifa, S., Chapman, A., Lakshmanan, L., Nierman, A., Paparizos, S., Patel, J., Srivastava, D., Wiwatwattana, N., Wu, Y., & Yu, C. (2002, December). TIMBER: A native XML database. *The VLDB Journal, The International Journal on Very Large Data Bases, 11,* 274-291.

OMG. (2001). *OMG meta-object facility (MOF).* Retrieved January 11, 2002, from *www.omg.org*

OMG. (2002). *MOF 2.0 Query/views/transformations RFP*. Retrieved February 10, 2004, from *www.omg.org*

OMG. (2003). *OMG XML metadata interchange (XMI) specification*. Retrieved March 3, 2002, from *www.omg.org*

OMG Architecture Board. (2001). Model driven architecture: A technical perspective. Retrieved January 7, 2001, from *www.omg.org*

W3C. (2003, August). *XQuery 1.0: An XML query language (working draft)*. Available at *http://www.w3.org/TR/xquery/*

Warmer, J., & Kleppe, A. (1998). *The object constraint language: Precise modeling with UML*. Boston: Addison-Wesley.

Chapter IV

Support for Collaborative Component-Based Software Engineering

Cornelia Boldyreff, University of Lincoln, UK

David Nutter, University of Lincoln, UK

Stephen Rank, University of Lincoln, UK

Phyo Kyaw, University of Durham, UK

Janet Lavery, University of Durham, UK

Abstract

Collaborative system composition during design has been poorly supported by traditional CASE tools (which have usually concentrated on supporting individual projects) and almost exclusively focused on static composition. Little support for maintaining large distributed collections of heterogeneous software components across a number of projects has been developed. The CoDEEDS project addresses the collaborative determination, elaboration, and evolution of design spaces that describe both static and dynamic compositions of software components from sources such as component libraries, software service directories, and reuse repositories. The GENESIS project has focussed, in the development of OSCAR, on the creation and maintenance of large software artefact repositories. The most recent extensions are explicitly

addressing the provision of cross-project global views of large software collections and historical views of individual artefacts within a collection. The long-term benefits of such support can only be realised if OSCAR and CoDEEDS are widely adopted and steps to facilitate this are described.

Introduction

The systemic representation and organisation of software descriptions (e.g., specifications, designs, interfaces, and implementations) of large distributed applications using heterogeneous software components have been addressed by research in the Practitioner and AMES projects (Boldyreff et al. , 1990; Boldyreff, 1992; Boldyreff, Burd, Hather, Mortimer, Munro, & Younger, 1995; Boldyreff, Burd, Hather, Munro, & Younger, 1996). The Practitioner project explicitly addressed the reuse of software concepts and developed a standard form to handle representations of software concepts from their specification to their associated implementations as components. The AMES project, while focused on maintenance support, organised the associated software components at various levels of abstract representations using hypertext and the Web. In both projects, it was assumed that the underlying collections of software components would support software reuse and the subsequent evolutions of systems composed from components. However, without appropriate representations and organisations, large collections of existing software are not amenable to the activities of software reuse and software maintenance; these activities are likely to be severely hindered by the difficulties of understanding the software applications and their associated components. In both of these projects, static analysis of source code and other development artefacts, where available, and subsequent application of reverse engineering techniques were successfully used to develop a more comprehensive understanding of the software applications under study (Zhang & Boldyreff, 1990; Fyson & Boldyreff, 1998). Later research addressed the maintenance of a Web-based component library in the context of component-based software product line development and maintenance (Kwon, Boldyreff, & Munro, 1997). The classic horizontal and vertical software decompositions proposed by Goguen (1986) have influenced all of this research. While they are adequate for static composition, they fail to address the dynamic aspects of composing large distributed software applications from components especially where these include software services that may be dynamically bound at run-time.

Recent research within the CoDEEDS project has made some progress toward the determination of design spaces to support both the static and dynamic system composition as well as the determination of the physical deployment and long-term operation of large distributed systems composed from heterogeneous components (Boldyreff, Kyaw, Nutter, & Rank, 2003). The current prototype implementation of collaborative support for the determination, elaboration, and evolution of design spaces, based on the CoDEEDS framework (Boldyreff & Kyaw, 2003), employs as its base another development of our recent research within the GENESIS project, the Open Source Component Artefact Repository, OSCAR (Boldyreff, Nutter, & Rank, 2002a; Boldyreff, Nutter, & Rank, 2002b; Boldyreff, Nutter, & Rank, 2002c; Nutter, Boldyreff, & Rank, 2003).

The GENESIS project developed a generalised environment for process management in collaborative software engineering (Gaeta & Ritrovato, 2002). A key component of this environment is an underlying distributed repository, OSCAR, to hold the software artefacts (both the artefact data and its associated metadata). A software artefact is any component of a work product resulting from the software engineering process. Thus the support provided covers not only the engineering of software systems from reusable software components, but also more generic reuse based on any work product components, such as project plans, requirements specifications, designs, test cases, and so on.

The research areas addressed in this chapter are:

- Process-aware support for collaborative software engineering
- Management of software (and other) artefacts within and across software engineering projects
- Use of XML-based artefact representations and interchange formats

The remainder of this chapter is organised as follows: First, the background related to Web-based collaborative software development and software evolution are examined. Then, the overall design of OSCAR and the support for co-operative software development that it currently offers combined with CoDEEDS are described, along with extensions to OSCAR to provide historical awareness of artefact development across projects (Nutter & Boldyreff, 2003), and a global view of a number of distributed artefact repositories are elaborated. Finally, planned deployment and future research activities are discussed.

Web-Based Collaborative Software Development

The Web and its associated technologies facilitate communication and cooperation amongst software developers, enabling large collaborative software development projects to be undertaken. The open source community provides many examples of such projects. Multinational software projects also are commonplace within industry today. Various solutions are available to address the immediate support of these collaborative development projects throughout the life cycle of the project. These solutions, open source and commercial, vary considerably in the elements of collaborative development and project management they address. SourceForge, in the open source domain, provides basic support for managing cooperative development of software artefacts such as handling mailing lists, forums, repository services, and bug tracking. However, it does not support workflow, resource management, or collaborative work by many users on a single artefact (apart from the use of a CVS (Concurrent Versions System) repository to handle configuration management). Microsoft Project Professional supports enterprise project management over single or distributed sites in the commercial domain. It

concentrates on the workflow and planning elements of cooperative development but has no specific focus on software engineering projects, unlike Rational's range of products, which support industrial software development across a global enterprise in the commercial domain. There also are general, not software-development-specific, Web-based solutions that have been used to support cooperative working of distributed software development teams, such as SiteScape, which handles a central repository, with forum-like facilities for interaction, and Basic Support for Cooperative Work (BSCW), which formed the basis of the SEGWorld development (Drummond & Boldyreff, 1999). The GENESIS project has employed SiteScape to manage the deliverables associated with its various work packages and to coordinate document reviewing associated with the project's research and software developments. All of these current solutions support Web-based access to project-related data and artefacts under production by the software team.

In contrast, the OPHELIA (Dewar et al., 2002; Wilcox et al., 2002) project offers support for collaborative work using software tools and employs a CORBA-based tool integration approach to do this. Various tools including project planning (MS Project) and development tools (such as ArgoUML) have been integrated using the ORPHEUS implementation of the OPHELIA framework. These applications can interchange data with other modules of the ORPHEUS system, which perform tasks such as metrics calculation, recording traceability information, and archiving data. Theoretically, any tool may be integrated within ORPHEUS but providing truly universal data interchange is of course difficult and the effort required to integrate a tool is significant.

The use of standard representations such as UML and XML-based notations has a beneficial effect on the cost and efficiency of software engineering projects. The GENESIS and CoDEEDS projects represent artefacts as XML documents. This has allowed the rapid development of sophisticated tools to handle artefacts; using currently available tools for XML handling has avoided the requirement to build entirely new artefact handling software. The use of UML as a common communication language amongst software engineers is supported by both projects. UML improves communication at the human level, while use of an XML exchange format can facilitiate the exchange of software artefacts.

Current solutions lack any means of obtaining a global view of project data and software artefacts across a number of projects irrespective of the initial methods and tools employed during the project's lifetime. Here the underlying artefact management system, OSCAR, being developed within the GENESIS project and coupled with the CoDEEDS framework, offers the basis for delivering such support in the future. One benefit of this is that a collection of artefacts can be treated as a repository of reusable components. The navigation and search facilities provided by GENISOM support the discovery of reuse candidates.

The GENESIS platform offers a process-aware environment which supports distributed software engineering, allowing flexible modeling and control of a collaborative software engineering project. While the GENESIS platform is based around process modeling and control, CoDEEDS specifically supports software engineering, providing support for specific software-related tasks such as architectural design. Both projects address supporting the evolution of software artefacts during their development and subsequent deployments within a variety of systems.

Software Evolution

Boldyreff was one of the first to recognise the role of evolution within the process of engineering computer systems (Boldyreff, 1954). He distinguished between mathematical models of systems and their corresponding physical realisation, and noted the necessity to evolve these models in step. In the 1970s, Lientz and Swanson (1980) studied a large number of software projects in many data-processing organisations. The study showed that software maintenance consumed approximately half the time of software professionals in the organisations which responded to their questionnaire. Generally, larger organisations spent a larger proportion of their time on maintenance, though results varied across industries. Their study showed that in organisations where maintenance was considered as a separate activity, it consumes a smaller proportion of effort. Lientz and Swanson's study was carried out in the late 1970s, and the level of technology that was used by the organisations reflected this. For example, change logs were handled manually, and implementation languages such as COBOL and FORTRAN were common. Lientz and Swanson concluded, unsurprisingly, that larger and older systems have greater maintenance problems than smaller and newer systems, and that personnel issues, such as the skill level and turnover of staff, are of importance in determining the quality and effort of system maintenance.

Lehman and Belady (1985a) made a detailed study of the development of a single software system. In contrast to the method used by Lientz and Swanson, Lehman and Belady studied the software product (IBM's OS/360), rather than the organisation. They examined the system's size at each release point, and showed that the size (in terms of lines of code and number of modules) and complexity of a system grows with each successive release, unless specific effort is made to reduce these factors. During this work, Lehman and Belady developed the idea of software system types, using the terms S-type, P-type, and E-type to describe the three types (Lehman & Belady, 1985b).

S-type programs are the simplest kind, being those programs which are formally defined as a function between input and output, with no reliance on or interaction with their environment, such as simple UNIX software tools (e.g., grep and awk). P-type programs are those which solve real-world problems, and must use heuristics to arrive at approximate solutions. Examples include weather forecasting and chess playing, where the input to the software is well-defined and well-formed, but in order to arrive at a useful solution in a reasonable amount of time, approximations must be used. E-type software is the most complex and most interesting kind of software. An E-type program is situated in and interacts with its environment, leading to feedback between the software and the "real world." Total correctness of an E-type system cannot be shown in the abstract. Such software interacts with its environment, and thus it can be only be shown to be effective in a particular, given, situation.

The results of these studies motivated Lehman to develop his laws of software evolution (Lehman, 1979; Lehman, 1996; Lehman, Ramil, Wernick, Perry, & Turski, 1997). These laws describe the behaviour of software systems over time (Lehman, 1996). They are:

- **Continuing Change:** An E-type program must either adapt or become obsolescent.
- **Increasing Complexity:** Unless an evolving program has work done specifically to reduce its complexity, it will become more complex as a result of the evolution.
- **Self-Regulation:** The evolution process is self-regulating, with statistically determinable trends and invariants.
- **Invariant Work-Rate:** The average effective global activity rate is constant over the lifetime of the system.
- **Conservation of Familiarity:** The content of successive releases is statistically invariant.
- **Continuing Growth:** Functional content of a system must increase with each release in order to satisfy user demands.
- **Declining Quality:** Unless an E-type program is rigorously maintained and updated to its changing environment, it will be perceived as declining in quality.
- **Feedback System:** The evolution process for E-type programs is multi-loop and multi-level. Successful management of the process depends on recognising and accounting for this fact.

Two of the key problems of maintenance are understanding the software in order to determine where to make changes, and validating the changed version of a system — determining that the correct changes and no others have been made (Baxter & Pidgeon, 1997). One important cause of the difficulty of maintenance is the complexity of software systems (Jackson, 1998); understanding a system in its entirety is often necessary before even a simple change can be made and validated, thus the need for support environments to capture and preserve the developer's understanding of programs.

As previously described, there have been several studies of the evolution of software systems. These and other studies have led to models of the process and products of software evolution which have been used to manage and control software evolution. Process models identify the mechanism by which the evolution is carried out, and product models identify the characteristics of the software which are important with respect to evolution.

There are two complementary research approaches to software evolution. The first approach, related to reverse engineering, aims to devise methods of working with legacy systems, while the second approach, related to forward engineering, attempts to design software that is easy to change. Whether a software system has been designed for ease of modification or not, there are common tasks which must be performed. In order to change a software system, the software engineer performing the task must understand both the system and the changes to be made (Takang & Grub, 1996). The software engineer must be able to verify that exactly the required changes have been made to the software.

Various techniques for handling software evolution have been described in the literature, including those by Takang and Grub (1996) and Pigoski (1996). Takang and Grub describe

several software life cycle processes, and put each in the context of evolving software systems, while Pigoski takes a more evolution-centred approach, concentrating more on the processes which occur after the development of a software system. Pigoski describes software evolution processes, metrics, and management issues.

While developing software which is easy to change is not entirely removed from changing so-called "legacy" software, it is sufficiently different to merit separate treatment. Various techniques for creating software have been described. These range from product-oriented guidelines for developing understandable source code (McConnell, 1993; Kernighan & Pike, 1999) to processes with attempts at psychological grounding in program comprehension (Smith, 1999).

There have been several attempts to categorise methods for dynamically changing software at run-time. These include simple techniques based on plug-ins (i.e., dynamically loadable modules) and parameter alteration (Rubini, 1997), and more sophisticated approaches based on component replacement or adaption (Bihari & Schwan, 1991; Segal & Frieder, 1989).

The lack of explicit representation of communication in a software system causes problems with the evolution of the system; communication is a key part of a software system and should be explicitly represented rather than implicitly inferred. Maintaining the existence of connectors through to the run-time instantiation of the code allows connectors to encapsulate more information about the communication that occurs between components, to contribute to the mobility, distribution, and extensibility of systems, and to act as domain translators, providing mappings from messages in one format to messages in another (Oreizy, Rosenblum, & Taylor, 1998).

The initial design of a modern system usually aims to have low inter-component coupling. This coupling between modules increases as a system is maintained (Lehman, 1998). Whatever the initial architecture of a software system, maintenance of the system without regard to the effects on the architecture will cause degradation of architecture (Lehman, 1996). There are several ways to tackle the problems here:

- Use a process of maintenance that pays explicit and careful attention to the architecture of the system.
- Design the architecture of the system in such a way that maintenance can be carried out in a way that preserves the structure and 'cleanliness' of the system.

When building a software system of significant size, reuse of existing pieces of software is desirable. Usually, unless the components have been specifically designed to work together and do not violate each others' assumptions, simple composition of components is not possible. Each component will make different assumptions about the environment and the behaviour of other components in the system, leading to so-called architectural mismatch (Garlan, Allen, & Ockerbloom, 1995). The most common approach to tackling this mismatch is to "wrap" components (commonly by inserting "glue" code between them) to insulate them from each other and to transform the input and output (Shaw, 1995).

One approach to architectural reuse is the concept of product-line architectures. These provide the opportunity to reuse parts of previously existing systems in later software, though this requires a significant amount of work to achieve and is hard to perform after-the-fact (Bosch, 1999).

Use of the C2 architectural style (Oreizy et al., 1998), which is based on a layered system of components and connectors, has been claimed to ease run-time software evolution; evolution without re-compilation of the system, in such a way that the system retains its integrity without becoming successively brittle over modifications (Oreizy & Medvidovic, 1998). Two types of system change are identified: changes to the system requirements, and changes to the implementation that do not affect the requirements.

Work on run-time architectural evolution has, in general, concentrated on providing the ability to dynamically replace components. This typically requires provision to be made at design time (Amdor, de Vicente, & Alons, 1991; Oreizy, 1998).

Distributed systems offer further challenges and opportunities. Large distributed (and other) systems may need to remain functional for long periods of time without interruption. In order to tackle this, Kramer and Magee (1985) propose replacing traditional (build-time) static configuration with incremental dynamic reconfiguration. This requires a greater separation between programming (implementation of behaviour) and configuration (implementation of composition), and requires a configuration language distinct from the programming language(s) used in the system. The more recent C2 architectural style advocates explicit representation of connectors, which provides the ability to abstract away from distribution and to insulate components from changes occurring in other parts of the system (Oreizy & Taylor, 1998).

There are several approaches to handling the evolution of a software system. These fall into the two categories of process-oriented solutions and product-oriented solutions. The GENESIS platform supports process-oriented software evolution, while the CoDEEDS project aims to assist with the maintenance of knowledge about software engineering products that have been developed collaboratively.

Suppport for Collaborative Development

In order to realise the approaches and models in practice, software engineering support environments with explicit provision for evolutionary design of component-based systems are required. Two complementary projects are described in greater detail.

The CoDEEDS Project

The CoDEEDS project is concerned with the Collaborative Determination Elaboration and Evolution of Design Spaces (Boldyreff, Kyaw, Nutter, & Rank, 2003b; Boldyreff & Kyaw, 2003). It provides support to design teams enabling them to record their determi-

nation of the solution space in the development of large complex distributed systems composed of heterogeneous software components. The result is a potentially N-dimensional design space layered by static and dynamic views of the component subsystems and models of their deployed instances within the system being designed and deployed in practice. The design environment being developed as part of the CoDEEDS project supports collaborative design throughout the system life cycle with an agent-based architecture to support the design team in their various activities.

Different members of the design team may employ their own preferred design methods and tools when carrying out the detailed design work. The CoDEEDS environment provides a global view of the overall design of the system and the various design decisions that have been made in its composition from a number of potentially heterogeneous components. Figure 1 indicates the primary areas (use cases) supported by the GENESIS and CoDEEDS systems; it shows both the overlapping and discrete primary areas addressed by each system.

The GENESIS Project and OSCAR

The GENESIS project is focused on the development of a Generalised Environment for Process Management in Co-operative Software Engineering. In the context of Figure 1 it addresses the needs for process and work product management. It is employed at both the project management and process workflow level. It complements the design rationale capture of the CoDEEDS system through its support of process engineering and collaborative activity recording. The GENESIS project has developed a low-overhead platform to support collaborative software engineering. The system has been designed to be process *aware,* but *nonintrusive*; like CoDEEDS, it does not mandate methods and tools to be employed by the development team. GENESIS is now an open source project that was seeded by initial closed-source developments by the project partners.

Figure 1. GENESIS and CoDEEDS overlap

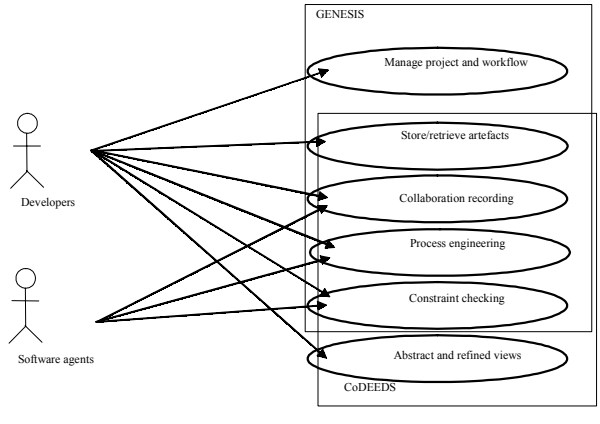
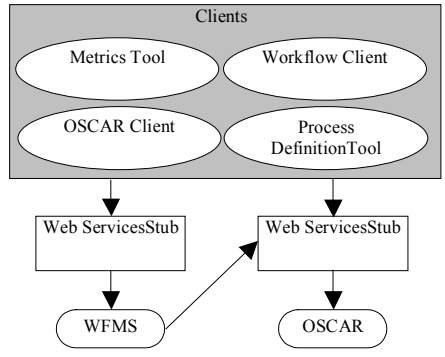


Figure 2. Overview of GENESIS



GENESIS, outlined in Figure 2, provides a solution for modeling and enacting workflow processes, and for managing both planned and unplanned work products. The process enactment is distributed over multiple physical sites coordinated by a global process at one site. Both local and global processes are managed via the GENESIS workflow management system.

Underlying both the GENESIS platform and the CoDEEDS system is an artefact management system, OSCAR, which acts as a repository for all artefacts resulting from development. OSCAR supports the creation, storage, retrieval, and presentation of artefact data elements and their associated metadata. "Everything is an artefact" is the view of the repository's data; this results in a simplified data model throughout OSCAR. By using Castor, an open source data-binding framework, in the implementation of OSCAR, the ability to treat artefacts as objects and documents simultaneously has been achieved allowing for flexible processing and extension of artefacts and their associated types. The actual storage of the artefact content is achieved through plug-ins to external storage mechanisms such as CVS. An abstraction over software configuration management (SCM) is currently mapped to a CVS plug-in and a plug-in for the Perforce SCM system is underdevelopment. Similarly plug-ins for searching are possible, such as the GENISOM extension described in the next section. Instrumentation to collect data about the users and system activities provides the basis for awareness extensions also described in the next section, and potentially for studies of collaborative working in the future.

Currently OSCAR is shipped with the following set of basic artefact types:

- **Software –** specifications, designs, code, and so forth
- **Annotation –** any additional information such as e-mail messages and other discussion that may help users of the original artefact
- **Human Resource –** description of the relevant software engineering personnel
- **Project –** workflow models and enactment descriptions
- **Default –** all artefacts are extensions of this

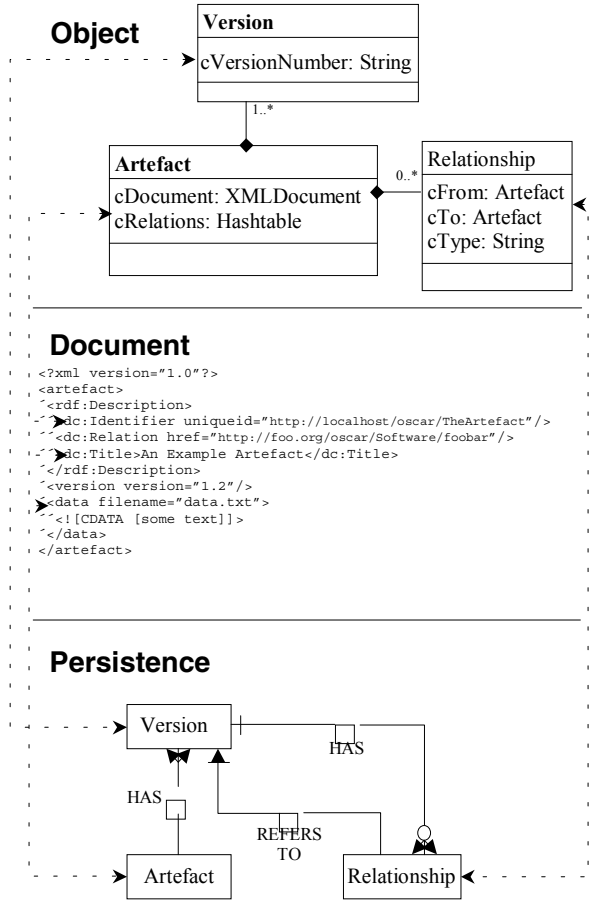
The user may extend this default set of types, at present new types may only be added to the system when the server is started. In particular, the CoDEEDs and GENISOM projects expanded the set of artefact types for their own purposes.

OSCAR's restrictive RMI interface is being complemented with a more accessible Web Services interface to ease deployment of the system in user environments where access through a firewall is necessary. This alternative interface will be useful to industrial users of OSCAR and to users in Open Source projects (Boldyreff, Lavery, Nutter, & Rank, 2003a).

Extending Artefact Types

To extend the set of types, two things are required: a set of Java classes derived from the base artefact type containing the functionality provided by the new type, and a Castor mapping file to translate between instances of the class and an XML document. Once the

Figure 3. Artefact representation



new type has been written and tested, the OSCAR server must be reconfigured and restarted to recognise the new type. Users may then create and modify instances of the type like any of the basic types.

There is a faster way to extend the set of "types": that is to use the default classes and mapping file but under a new name. Obviously, the user gains no new features by doing this but can differentiate a set of otherwise similar artefacts by changing the type name without needing to spend time developing new classes. For example, the CoDEEDS project first used the default artefact type under the name "CoDeedsArtefact" before writing classes and mappings for an artefact that provided features necessary for the project, which then replaced the default type.

Figure 3 illustrates how the XML-based representation of artefacts forms a link between the higher level human-understandable representation, which is rendered as a Java object describable in UML, and the lower level database (entity-relationship) representation, which is used to provide persistence.

Basic Artefact Operations

Currently high-level artefact operations exist for automatic indexing to support search and retrieval, and for various transformations to allow for flexible presentation of artefacts to users, usually as an XML document, sometimes as an object. Also, basic facilities common to all artefacts exist, including the ability to query and modify the basic metadata, store data within an artefact, store and retrieve versions of an artefact or collection of artefacts, and make relationships between artefacts.

Extending Oscar with Addtional Repository Services

We describe two additional services that are part of OSCAR alongside the basic management facilities described previously.

Historical Awareness

The possibility of extending OSCAR with historical awareness arises along with the cross project historical data that is captured as OSCAR is used to support a number of projects and as data sharing between distributed OSCARs is realised.

Historical awareness deals with a collection of heterogeneous artefacts, allowing the user to view the complete context of an artefact's creation and history of changes into its present form across a number of projects, rather than a contextless view of changes to a single project artefact (Nutter & Boldyreff, 2003). Historical awareness is superficially similar to change logs and history views provided by SCM systems but, unlike

these systems, provides information that has not been explicitly requested by the user. One way of displaying historical data is via a timeline relating the changes made to an artefact by various users over time. Such a display can be driven by events as they occur providing immediate feedback to developers sharing an artefact across projects or within a single project. In effect, through historical awareness, users gain a view of the software artefact's evolution over time and across a number of uses within various projects.

The implications of supporting component reuse via this feature are that historical awareness may be able to provide potential users of the component with the big picture of the component's development over time necessary for program comprehension, which must precede effective reuse and evolution. It also gives them immediate feedback from other developers reusing the component and possibly adapting or evolving its functionality, thus preventing conflict (Nutter & Boldyreff, 2003).

GENISOM

A prominent problem within the field of Component-Based Software Engineering concerns finding suitable components to reuse. Reusable assets are in abundance over the Web and in libraries, but it is extremely difficult to locate reusable software components that are relevant to a particular application. The necessary organisation is often lacking and difficult to achieve given the dynamic nature of such software collections. This problem also can be found where a large evolving software system consists of an ever-growing number of components and the management, and hence the comprehension of the associated software components tends to become increasingly difficult. In the GENISOM project, we have applied self-organising maps (SOMs) to a large population of software components and developed various visualisations of the SOMs. Their effectiveness in relation to the organisation of a large software collection and their usage by software engineers wishing to search the collection has been investigated (Brittle, 2003; Brittle & Boldyreff, 2003).

SOMs are an adaptive technique used to hierarchically (in our case) organise a large search space into a two-dimensional array of smaller spaces. The organisation is performed using an unsupervised neural network (Kohonen et al., 2000).

OSCAR's initial large-scale population for demonstration purposes is derived from the packages of the Debian open source project and consists of just over 1,500 software artefacts. This population with its extracted metadata has been employed in some experimental studies to gauge the effectiveness of using SOMs to classify large collections of software artefacts in the GENISOM project (Brittle, 2003). In GENISOM, we have replicated Kohonen's original WebSOM (Kohonen et al., 2000) and extended it to the domain of Web-based software artefact collections. SOMs are used as a data visualisation technique to support users browsing and searching large collections of data by representing the collection's population as an interactive map, thereby exploiting computer technology and people's abilities to comprehend visual representations. Even though reusable assets are in abundance, a growing problem is the ability to actually locate assets that are relevant for reuse. Organisation of a collection is therefore a necessity, and the GENISOM project and other research (Merkl, 1998) have come to the conclusion that SOMs are viable organisational tools that could be used instead of

hierarchical or faceted classification. SOMs also provide a virtually automatic organisation process that could save on the costs associated with employing reuse librarians and reduce the amount of time needed to train engineers in the use of the library. More recently, GENISOM has been redeveloped to provide a front-end to OSCAR, and the test population has been expanded to include the Java software components that comprise the current implementation of OSCAR (Brittle & Boldyreff, 2003).

The GENISOM maps provide potential component reusers with various views of the software collection. Figure 4 illustrates one view of such a map.

Our preliminary results, applying a prototype implementation GENISOM to the Debian and OSCAR components, show promise and support our belief that SOMs are an ideal solution to organising the incrementally expanding content of the large distributed repositories that we anticipate will result from OSCAR's usage by a growing number of software development projects.

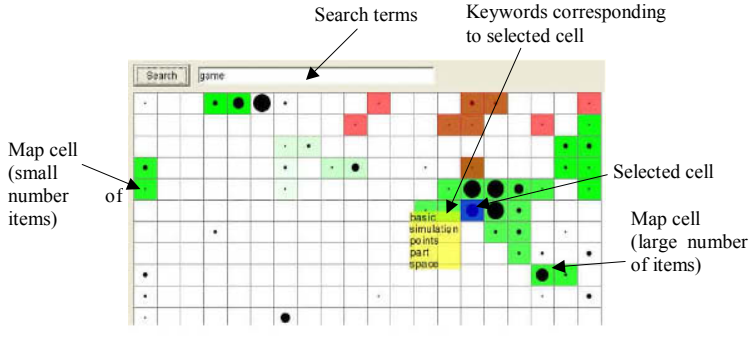
Extending OSCAR for GENISOM and Awareness

Extending OSCAR to support both these projects will require modification of both the client and server parts of OSCAR. Though the goals were different, some of the architectural modifications are similar.

GENISOM at first required client-side modifications to generate useful maps from a user's own collection of artefacts in a workspace. These initial modifications required the addition of a new artefact type to describe a particular SOM configuration and a special artefact type describing Debian package metadata used to represent the test artefact population.

The modifications to the client entailed adding a new user view in addition to the existing hierarchical view of the workspace contents, and allowing the user to switch between the views at will. Dialogues to guide the user through the process of creating a self-organising map of the contents of their workspace (and a descriptive artefact) were prepared and added to the client. A tool to extract test artefacts from the Debian packages file was prepared.

Figure 4. GENESOM 2D map view



Adding awareness support requires modification of the client Though in contrast to the view added for GENISOM, this will not allow navigation of the complete workspace, just the parts of the workspace which are affected by the activities of other software engineers. Several server-side modifications are necessary to deliver awareness information to the client; an event handler is required to convert change and dependent change events generated by artefacts into a form suitable for display in the awareness view. This handler will feed the information it creates to the distribution mechanism, which communicates with the peers in a distributed awareness network.

The awareness network is built by closely linking clients (few hops) working with similar artefact collections; potential algorithms for doing this are described in previous work (Nutter & Boldyreff, 2003). Awareness information messages are then given a time to live (TTL) and sent to the originating client's immediate peers. From there, messages are propagated further until the TTL has expired. Since nearby peers will all be using similar artefacts, this approach will ensure that information expires once it becomes irrelevant, keeping the network clear of spurious traffic and removing the need to filter information for relevance on the client.

This method necessarily means that some information will be lost when the network is imperfectly arranged as it will not reach all the clients interested in it. However, the display method outlined for historical awareness can cope with lost information.

Deployment

Within the framework of the GENESIS project, the consortium's industrial partners have deployed the GENESIS platform including OSCAR in a number of user trials. Members of the GENESIS project team have used it to support their own internal development. A stable version of the GENESIS platform is available on SourceForge (http://sourceforge.net/projects/genesis-ist). The CoDEEDS system is currently a research prototype which is being prepared for release as an open-source system.

The GENESIS platform has been evaluated in the industrial partners' organisations (LogicDIS and Schlumberger) using a comprehensive test bed. In each partner's organisation, the platform was used to model an already completed project. The project was re-run with the assistance of the GENESIS platform.

Consideration has been given to the adoption of the GENESIS platform by organisations. For large organisations with highly distributed cooperating teams, the adoption of a new technology is a complex process that requires an organisation to consider the technology in context of the organisation's business goals (Lavery, Boldyreff, Nutter, & Rank, 2003). Prior to the adoption of GENESIS a large organisation must determine the answers to two difficult questions: Do the existing software processes require additional or improved technical support supplied by GENESIS?

Does the organisation need to improve their software processes and will GENESIS support that improvement effort?

It is essential to any organisation that the adoption of any new technology is based on the determined needs of the organisation. In the GENESIS project, we advocate the use of the Carnegie Mellon Software Engineering Institute's Capability Maturity Model (SW-CMM) (Dewar et al., 2002) to determine those organisational needs and to support an incremental technology adoption strategy (Lavery et al., 2003).

GENESIS and CoDEEDS are a collection of distinct systems that work together to provide effective support for the management of both software product evolution and software processes enactment. Thus, it is possible to introduce the individual systems incrementally based on the determined needs of the organisation.

To ease adoption of the platform, a stand-alone version of OSCAR has been developed and made available. As well as the tools described earlier to up-load the Debian project software, a simple import tool for Java software and other miscellaneous files has been developed. This has enabled the GENESIS project software to be easily transferred into OSCAR as part of the project's own use of its developments.

As with the local and global work processes, the work products managed by OSCAR will soon be visible in a similarly global name-space composed of multiple local OSCAR repositories. Also in progress for OSCAR is user-transparent metadata extraction and indexing functionality.

It is only with the widespread adoption of OSCAR and the development of much larger collections of software artefacts stored in OSCAR that advantages, such as being able to obtain global views of such collections held in distributed repositories, will become apparent.

Instrumenting the tools provided by both GENESIS and CoDEEDS will allow evolution studies of both software engineering processes and products to be performed. Monitoring the real behaviour of projects managed by the GENESIS workflow engine will allow studies of software development processes, indicating how closely real software engineering projects adhere to idealised models. Studying the evolution of products across a number of projects allows a full picture of the development effort to be obtained and may be the basis for predicting future changes.

The architecture of the GENESIS platform currently relies on the relatively tight binding of RMI. This is being transformed to a new architecture based on Web services. Once this has been done, the distribution model of the platform will be more flexible. It will no longer be necessary to maintain a strict one-to-one relationship between GENESIS and OSCAR installations; an instance of OSCAR could be shared by more than one GENESIS platform, or a single GENESIS project could use more than one repository.

The industrial partners have evaluated the GENESIS project in real projects. The feedback on the prototype platform that was evaluated has provided motivation for future development in terms of functionality, usability, and interaction mechanisms. The CoDEEDS prototype also is being released as an Open Source project. Feedback from its users will guide its further development.

Conclusion and Future Work

Our initial experimental developments show that GENISOM provides an effective way to organise a large collection of artefacts. Research is in progress to evaluate visualisation techniques applied to the associated SOMs in terms of their utility to support the software reuse by software engineering teams.

The applicability of collaborative technologies and theory to software engineering in the open source environment has not yet been studied. The CALIBRE Co-ordinated Action will provide an opportunity for collaboration experts and Open Source stakeholders to employ tools and techniques for collaboration in highly distributed projects.

We also have proposed a track of research complimentary to the UK E-Science agenda (Boldyreff & Nutter, 2003). The objective of this research programme is to study the needs of collaborators on the scientific grid who will be performing the following activities:

- Designing experiments, much like collaborative design of software
- Replicating or studying previous experiments (data provenance is therefore important)
- Collaborating on data analysis, requiring descriptions of scientists working on the system, data sources and full traceability between them

The eScience agenda itself is very technology focussed, concentrating on the development of technologies for distributed computing and data exchange. However, we believe that collaboration is at the heart and critical to the success of scientific endeavour and must be considered in any large-scale scientific system for that system to be successful.

This chapter has described two open source projects which support collaboration using UML and XML. Use of standard representation formats such as these plays a critical role in facilitating software reuse and the evolution of software artefacts. Support is needed for both the process of software engineering as well as the products of these processes. GENESIS provides support for the processes, OSCAR and CoDEEDS provide support for the products. As software engineering matures as a discipline, software reuse has become a more viable option and is becoming a more important part of the software engineer's toolkit. The systems described here support collaborative development, per se, and also collaboration across projects at different times, by supporting reuse, aided by common standard representations.

Acknowledgments

We wish to acknowledge and thank both James Brittle and Christopher Korhonen for their work with the GENESIS project team. GENESIS was funded by the EU under their IST programme, and CoDEEDS was funded by the UK EPSRC.

References

Amdor, J., de Vicente, B., & Alons, A. (1991). Dynamically replaceable software: A design method. *Proceedings of the Third European Software Engineering Conference, (ESEC)* (pp. 210-228).

Baxter, I. & Pidgeon, C. W. (1997). Software change through design maintenance. *Proceedings of the 1997 International Conference on Software Maintenance (ICSM 97)* (pp. 250-259).

Bihari, T. E. & Schwan, K. (1991). Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems, 9*(2), 143-174.

Boldyreff, A.W. (1954). *Systems engineering.* Technical Report P-537. Mathematics Division, The RAND Corporation, 16 June 1954. Available at *http://hemswell.lincoln.ac.uk/~cboldyreff/boldyreff-se.pdf*

Boldyreff, C. (1992). A design framework for software concepts in the domain of steel production. *Proceedings of the Third International Conference on Information System Developers Workbench,* Gdansk, Poland, September 22-24.

Boldyreff, C. & Kyaw, P. (2003). A framework for developing a design evolution environment. *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC).*

Boldyreff, C. & Nutter, D. (2003, Septmeber 8). Supporting collaborative grid application development within the e-science community. *First International Workshop on Web Based Collaboratories, collocated with IADIS WWW/Internet,* Carvoiero, Algarve.

Boldyreff, C., Burd, E.L., Hather, R.M., Mortimer, R.E., Munro, M., & Younger, E.J. (1995). The AMES approach to application understanding: A case study. *Proceedings of the International Conference on Software Maintenance.* IEEE Computer Press.

Boldyreff, C., Burd, E.L., Hather, R.M., Munro, M., & Younger, E.J. (1996). Greater understanding through maintainer driven traceability. *Proceedings of the 4th Workshop on Program Comprehension,* April (pp. 100-106). IEEE Computer Press.

Boldyreff, C., Elzer, P., Hall, P., Kaaber, U., Keilmann, J., & Witt, J. (1990). PRACTITIONER: Pragmatic support for the reuse of concepts in existing software. *Proceedings of Software Engineering 1990 (SE90),* Brighton, UK. Cambridge, UK: Cambridge University Press.

Boldyreff, C., Kyaw, P., Nutter, D., & Rank, S. (2003). Architectural framework for a collaborative design environment. *Proceedings of Second ASERC Workshop on Software Architecture,* Banff, Canada.

Boldyreff, C., Lavery, J., Nutter, D., & Rank, S. (2003). Open-source development processes and tools. *Proceedings of Taking Stock of the Bazaar: The Third Workshop on Open Source Software Engineering,* Portland, Oregon.

Boldyreff, C., Nutter, D., & Rank, S. (2002a). Open-source artefact management for distributed software engineering. *Proceedings of the Second Workshop on Open-*

Source Software Engineering at the 24th International Conference on Software Engineering.

Boldyreff, C., Nutter, D., & Rank, S. (2002b). Active artefact management for distributed software engineering. *Workshop on Cooperative Supports for Distributed Software Engineering Processes: Proceedings of the 26th IEEE Annual International Computer Software and Applications Conference (COMPSAC).*

Boldyreff, C., Nutter, D., & Rank, S. (2002c). Architectural requirements for an open source component and artefact repository system within GENESIS. *Proceedings of the Open Source Software Development Workshop,* Newcastle upon Tyne, UK, February 25-26 (pp. 176-196).

Bosch, J. (1999). Evolution and composition of reusable assets in product-line architectures: a case study. *Proceedings of the First Working IFIP Conference on Software Architecture* (pp. 321-340).

Brittle, J. (2003). *Self organizing maps applied to Web content.* Final Year Project Report, Department of Computer Science, University of Durham.

Brittle, J. & Boldyreff, C. (2003). Self-organising maps applied in visualising large software collections. *Proceedings of IEEE VISSOFT.*

Dewar, R. G., Mackinnon, L. M., Pooley, R. J., Smith, A. D., Smith, M. J., & Wilcox, P. A. (2002, September 13-15). The OPHELIA project: supporting software development in a distributed environment. *IADIS WWW/Internet 2002.*

Drummond, S. & Boldyreff, C. (1999). SEGWorld: A www-based infrastructure to support the development of shared software engineering artifacts. *Proceedings of the Workshop on Web-Based Infrastructures and Coordination Architectures for Collaborative Enterprises. IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)* (pp. 120-125). IEEE Press.

Fyson, M. J. & Boldyreff, C. (1998). Using application understanding to support impact analysis. *Journal of Software Maintenance: Research and Practice, 10* , 93-110.

Gaeta, M. & Ritrovato, P. (2002). Generalised environment for process management in cooperative software engineering. *The 26th Annual International Computer Software and Application Conference Proceedings* (pp. 1049-1053). IEEE.

Garlan, D., Allen, R., & Ockerbloom, J. (1995). Architectural mismatch or why it's hard to build systems out of existing parts. *Proceedings of the 17th International Conference on Software Engineering* (pp. 179-158).

Goguen, J.A. (1986). Reusing and interconnecting software components. *IEEE Computer*, February, 16-28. (Reprinted in Freeman, P. (Ed.). *Tutorial: Software reusability* (pp. 251-263). IEEE Computer Society Press.

Jackson, M. (1998). Will there ever be software engineering? *IEEE Software, 15*(1), 36-39.

Kernighan, B.W. & Pike, R. (1999). *The practice of programming.* Reading, MA: Addison Wesley Longman.

Kohonen, T., Kaski, S., Lagus, K., Salojarvi, J., Honkela J., Paatero, V., & Saarela, A. (2000). Self organization of a massive document collection. *IEEE Transactions on Neural Networks, 11*(3), 574-585.

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

Kwon, O. C., Boldyreff, C. & Munro, M. (1997). An integrated process model of software configuration management for reusable components. *Proceedings of the Ninth International Conference on Software Engineering & Knowledge Engineering (SEKE'97)*, Madrid, June 18-20.

Lavery, J., Boldyreff, C., Nutter, D., & Rank, S. (2003). *Incremental adoption strategy for the GENESIS platform.* GENESIS Project Report. University of Durham. Available at *http://www.dur.ac.uk/janet.lavery/documents/AdoptStratFinal.pdf*

Lehman, M.M. (1979). On understanding law, evolution and conservation in the large program life cycle. *Journal of Systems and Software, 1*, 213-221.

Lehman, M.M. (1996). Laws of software evolution revisited. In *Proceedings of EWSPT96, number 1149 in Lecture Notes in Computer Science* (pp. 108-124). Heidelberg, Germany: Springer-Verlag.

Lehman, M.M. (1998). Software's future: Managing evolution. *IEEE Software 15*(3), 40-44.

Lehman, M.M. & Belady, L.A. (1985a). *Program evolution: Processes of software change*. (Number 27 in APIC Studies in Data Processing.) Academic Press.

Lehman, M.M. & Belady, L. A. (1985b). Programs, life cycles and laws of software evolution. In *Program evolution: Processes of software change* (pp. 393-449). (Number 27 in APIC Studies in Data Processing.)

Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E., & Turski, W.M. (1997). Metrics and laws of software evolution: The nineties view. In K.E. Eman & N.H. Madhavji (Eds.), *Elements of software process assessment and improvement* (pp. 20-32). Albuquerque, NM: IEEE CS Press.

Lientz, B.P. & Swanson, E.B. (1980). *Software maintenance management: A study of the maintenance of computer application software in 487 data processing organizations.* Reading, MA: Addison-Wesley.

McConnell, S. (1993). *Code complete: A practical handbook of software construction.* Redmond, WA: Microsoft Press.

Merkl, D. (1998). Self-organizing maps and software reuse. In *Computational intelligence in software engineering*. World Scientific.

Nutter, D. & Boldyreff, C. (2003). Historical awareness support and its evaluation in collaborative software engineering. *Proceedings of the Workshop on Evaluation of Collaborative Information Systems and Support for Virtual Enterprises at the 12th IEEE international Workshops on Enabling Technologies for Collaborative Enterprises (WETICE).*

Nutter, D., Boldyreff, C., & Rank, S. (2003). An artefact repository to support distributed software engineering. *Proceedings of 2nd Workshop on Cooperative Support for Distributed Software Engineering Processes, CSSE 2003,* Benevento, Italy.

Oreizy, P. (1998). Issues in modeling and analyzing dynamic software architectures. *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis,* Marsala, Sicily, Italy.

Oreizy, P, & Medvidovic, M. (1998) Architecture-based runtime software evolution. *Proceedings of the International Conference on Software Engineering,* Kyoto, Japan (pp. 19-25).

Oreizy, P. & Taylor, R.N. (1998). On the role of software architectures in runtime system reconfiguration. *IEE Proceedings-Software, 145*(5), 137-145.

Oreizy, P., Rosenblum, D.S., & Taylor, R.N. (n.d.). *On the role of connectors in modelling and implementing software architectures.* Technical Report UCI-ICS-98-04. Department of Information and Computer Science, University of California, Irvine,

Paulk, M.C., Curtis, B., Chrissis, M.B., & Weber, C.V. (1993). The capability maturity model for software. *IEEE Software, 10*(4), 18-27.

Pigoski, T.M. (1996). *Practical software maintenance.* New York: John Wiley & Sons.

Rubini, A. (1997). The sysctl interface. *Linux Journal, 41.* Available at *http://www2.linuxjournal.com/lj-issues/issue41/2365.html*

Segal, M.E. & Frieder, O. (1989). Dynamic program updating: A software maintenance technique for minimizing software downtime. *Journal of Software Maintenance: Research and Practice, 1*(1), 59-79.

Shaw, M. (1995). Architectural issues in software reuse: it's not just the functionality, it's the packaging. *Proceedings of the IEEE Symposium on Software Reusability.*

Smith, D. D. (1999). *Designing maintainable software*. Springer-Verlag.

Takang, A.A. & Grub, P.A. (1996). *Software maintenance: Concepts and practice.* London: International Thomson Computer Press.

Wilcox, P.A., Smith, M.J., Smith, A.D., Pooley, R.J., MacKinnon, L.M., & Dewar, R.G. (2002). OPHELIA: An architecture to facilitate software engineering in a distributed environment. *The 15th International Conference on Software and Systems Engineering and Their Applications (ICSSEA),* Paris, December 3-5.

Zhang, J. & Boldyreff, C. (1990). Towards knowledge-based reverse engineering. *Proceedings of the Fifth Annual Knowledge-Based Software Assistant Conference,* Syracuse, New York, September 24-28.

Chapter V

Migration of Persistent Object Models Using XMI

Rainer Frömming, 4Soft GmbH, Germany

Andreas Rausch, Technische Universität Kaiserslautern, Germany

Abstract

Embrace the change! Change is a constant reality of software development, a reality that must be reflected in not only our software process but also our software production environment. With ever-changing customer requirements, modifications to the object model are required during software development as well as after product distribution. The associated migration of existing persistent object data is a nontrivial problem. This chapter presents the conceptualization and implementation of a tool for the automated migration of persistent object models. The migration is controlled by an XMI-based description of the difference between the old and the new object model. Both, the schema and the data of the persistent object model are migrated efficiently and reliably.

Introduction

"Time-to-market" is one of the major factors for success of software products today. Competitive pressure is forcing companies to introduce new versions of their software products with increased functionality to the market in ever-decreasing time intervals. For

instance, extreme programming (Beck, 2000) promises a development process to cope with this dramatically shortened life cycle.

However, agile software development with associated code-refactoring has its downsides as well. Short development cycles and fast assimilation to market requirements are possible, but this entails frequent enhancements and changes to the object model of the application under development. While changes to the object model and its corresponding program code are not necessarily crucial, due to modern development environments such as eclipse (Eclipse Foundation, n.d.), the migration of existing persistent object data in databases exposes the developer to supplementary, time-consuming tasks. Usually, there are two possibilities to address this problem: The first is to write additional program code besides the related product which handles the migration of data. The second is to include the logic how to access old and new data in the application itself. The drawback with the first possibility is that there is no framework which handles the basics of data migration to support the developer. This part can be very tricky, for example, without security features to backup your database or an architecture that can cope with class-naming-conflicts. When adapting the second possibility, the application is littered with code fragments only needed to differ between old and new data. This gets even worse when many new versions are built. Up to this point in time, no database vendor provides a suitable tool that addresses this migration problem in full complexity.

This results in the need for a tool which is capable in migrating the persistent object data from an aged software system to a new one in an automated, efficient, and reliable way. With the aid of such a tool, data migration expenses will be greatly reduced and the overall quality of the product will be improved considerably.

This chapter focuses on the concept and the functionality of such a tool, referred to here as ShapeShifter. The application domain of ShapeShifter is mainly concentrated on the object oriented software development cycle in conjunction with databases, where it improves data migration in terms of efficiency, traceability, and quality. ShapeShifter is written in Java, and in its present state, is able to migrate object-oriented databases from the Versant Corporation (n.d.). However, ShapeShifter is not basically limited to the Versant Database. Due to its flexible architecture and well designed interfaces, support for additional databases such as Oracle, DB2, or other relational databases is easily achievable and is already in preparation.

XMI: Describing Object Models

Any changes made to the object model entail knock-on changes to both the database schema and the persistent data. Although, small tools to support migration are delivered by the database vendors, these tools are insufficient and most of the migration work has to be done manually (Nierstrasz & Tsichritzis, 1995).

However, the migration process could be automated to a great extent, because most of the information you need is already at hand. The object models for both the new and old systems are available as source code and therefore also as a Unified Modeling Language (UML) description (Ambler, 2002; Fowler & Kendall, 2003).

Due to the fact that many different object-oriented languages and case tools exist, it would not be advisable for a tool developer to focus on a proprietary language specification or a special case tool file format. For further processing, a standardized, platform-independent representation of the object model should be used. The XML Metadata Interchange (XMI) standard adopted by the Object Management Group (OMG) (Grose, 2002) is particularly suitable for this. XMI provides a complete textual description of object-oriented models, which is in contrast to graphically description techniques like for the UML (Fowler & Kendall, 2003) more suitable for further automatic processing and transformation operations.

The main goal in the development of XMI was the creation of a vendor-independent industry standard for a model interchange format. Currently, all major CASE-Tools, as well as some other tools such as the XMI Toolkit of IBM Alphaworks (n.d.), can directly generate XMI from the existing source code or the UML description.

XMI: Describing Differences Between Object Models

XMI has a very important feature: It allows the description of the difference between two object models. This feature can be used to describe the migration of object models as a sequence of primitive operations on the object model. If the difference is "added" to the old version of the object model, the result will be the new one.

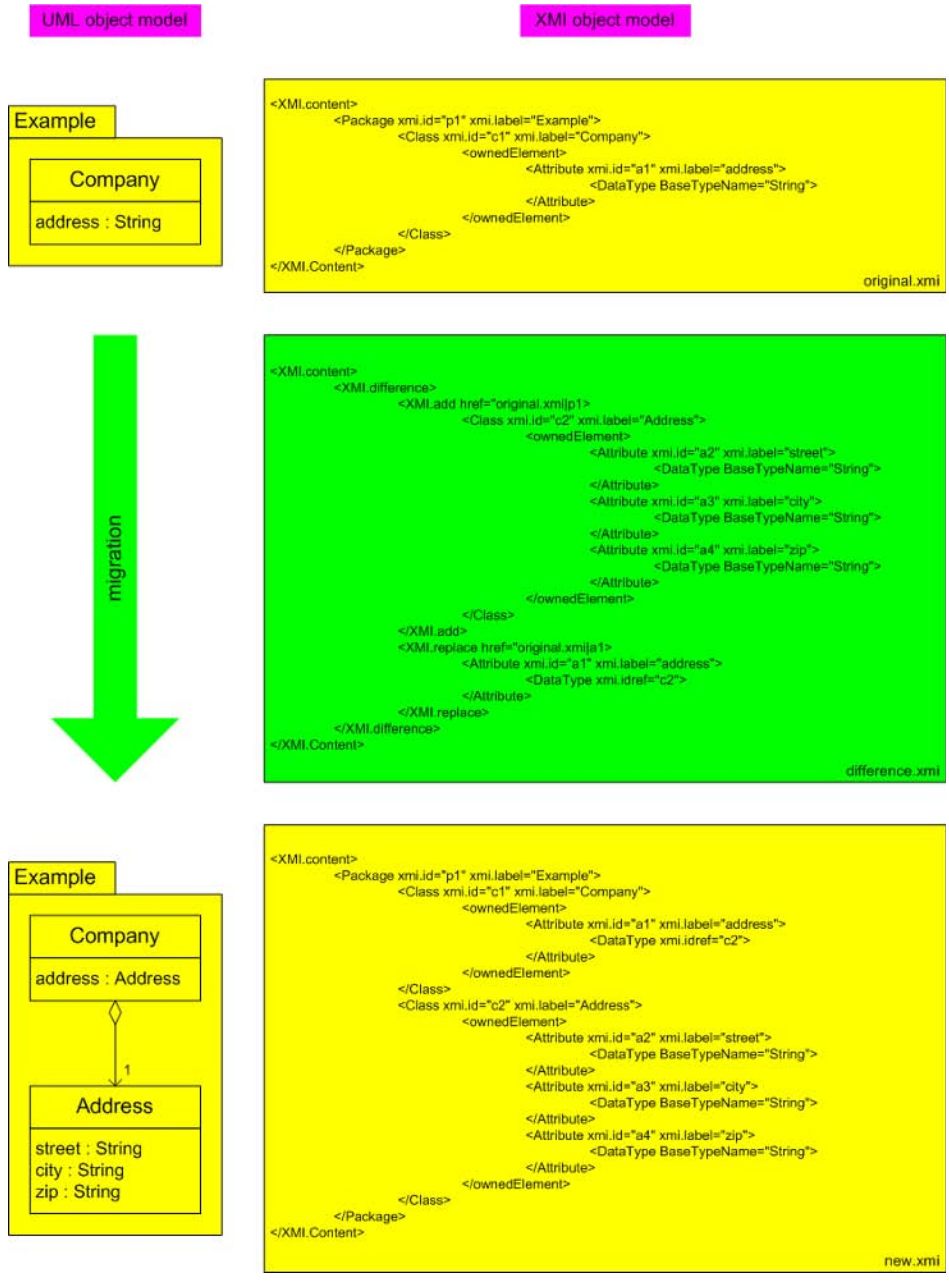
Figure 1 shows two versions of a simple object model (original.xmi, new.xmi), represented as an UML diagram (UML object model) and a XMI-based description (XMI object model). Additionally, the XMI-difference description can be seen in the middle of the illustration (difference.xmi).

At first it is important to understand the basic structure of a XMI description and how XMI works. The XMI standard defines three different types of operations: *XMI.add*, *XMI.delete,* and *XMI.replace*. These operations (or elements) are used to describe how a source model is changed into a target model. When the *XMI.difference* instructions are applied to a xmi model description, the model is converted into a target model. These basic operations add, delete, or change an attribute or a class in the object model. Every operation can hold plenty of attributes like *xmi.id*, *xmi.label*, *href,* and more. These attributes are basically used for naming and referencing elements in the object model and are needed to describe *where* changes have to be applied. Finally the tags *class* or *attribute* inside the operation tags describe the class and attribute modifications or additions in detail.

The example in Figure 1 uses the *XMI.add* operation to add a new class called *Address*, including all its attributes, to the model. The *XMI.replace* operation changes the type of the attribute *address* in the class *Company* from *String* to *Address*. These two operations are sufficient to describe the complete object model migration.

As mentioned, XMI descriptions of the object model can be generated very easily. Creating the XMI-Difference description is much more difficult, as this cannot be

Figure 1. Two simple object models represented as XMI



automated completely, with additional information from the developer needed. Formally spoken, this is an undecidable problem (Waller, 1991).

To explain this, we give a short example: There is no difference for the resulting schema when changing the type of the attribute *address* from *String* to *Address* through an *XMI.replace* on the address attribute or by deleting it with *XMI.delete* and inserting the new Address attribute with the *XMI.add* operation.

The end result of the whole data-migration process, however, would be totally different in these two cases. In the first case, the data must be converted from the *String* to the *Address* class, in the second case the data has to be deleted and the new Address attribute has to be initialized with a default value.

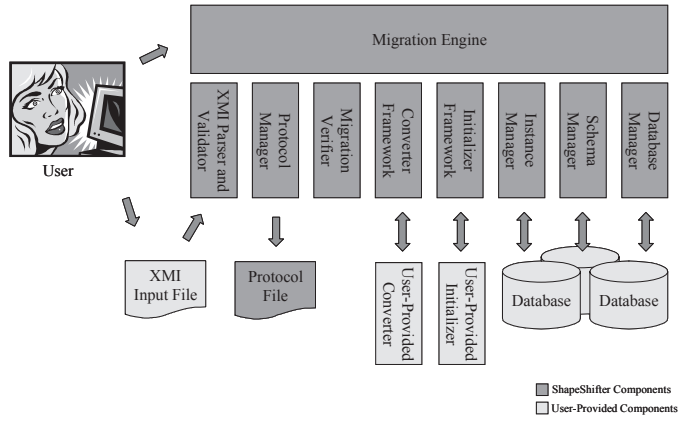
To sum up, tools like XMLDiff from IBM Alphaworks (n.d.), can only generate suggestions which still have to be reworked by the developer. If an XMI-Difference description is available, the schema migration can be done automatically. In the case of the data-migration though, this is generally not possible, as special converters and other additional information has to be prepared. This means that in most cases custom code has to be developed in order to migrate data with ShapeShifter.

Components of ShapeShifter

Figure 2 shows the main components and interfaces of ShapeShifter. The *Migration Engine* is the primary component of ShapeShifter which is used to manage the whole migration process. It takes control of the workflow of the migration and for this purpose it uses the offered services of the components below it as needed to perform all the tasks.

The *XMI Parser & Validator* processes the XMI input file and accepts only valid XMI files matching the respective Document Type Definition (DTD) (Harold & Means, 2002).

Figure 2. ShapeShifter components



The *Database Manager* handles all physical access operations like copying or deleting databases. The *Schema Manager* enables the *Migration Engine* to make changes to the object schema of the database. The *Instance Manager* provides read and write access on object instances.

For initialization of new classes that are created in the migration process, the *Initializer Framework* is used. The data migration is handled by the *Converter Framework*. These two frameworks provide standard converters and standard initializers for standard data types. For instance, the conversion of *int* to *long* is a standard conversion, whereas the migration of the custom attribute *address* from Figure 1 needs a custom converter. This can be included via the plug-in framework of ShapeShifter which will be explained later.

The *Migration Verifier* will check the database for consistency after the migration. The *Protocol Manager* documents the whole migration process.

Technical Architecture

Figure 3 shows an overview of the technical architecture of ShapeShifter. ShapeShifter uses three virtual machines which communicate via Remote Method Invocation (RMI) in order to perform the migration.

The first virtual machine (Java VM1) holds the *Migration Engine* as the primary component. In this process, the migration is controlled and coordinated. The second virtual machine (Java VM2) provides access to the original persistent object model. The third virtual machine provides access to the new persistent object model, plus it contains the *Schema Manager* in order to be able to make changes to the database schema.

Figure 3. ShapeShifter technical architecture

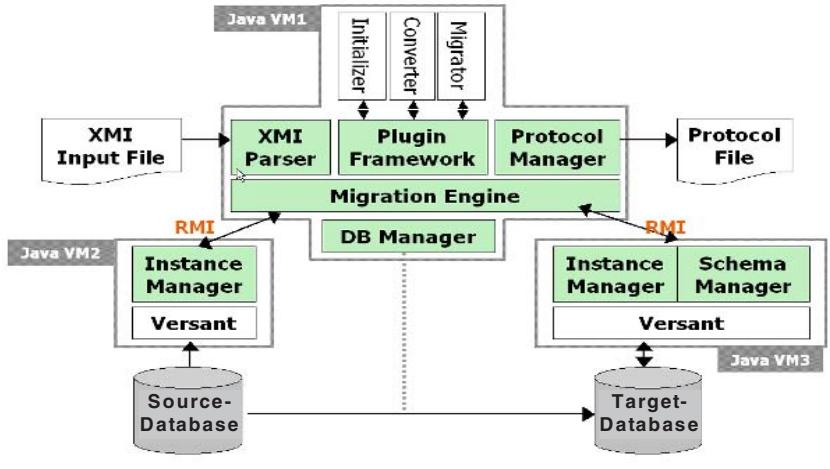
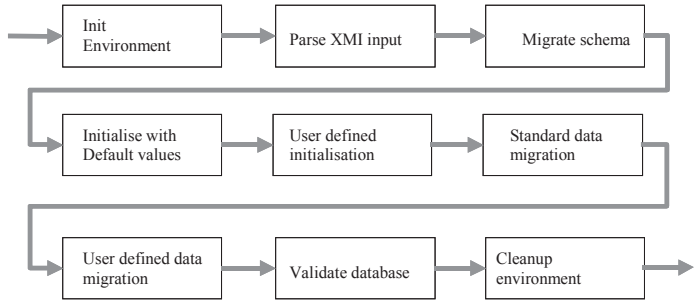


Figure 4. ShapeShifter procedure during the database migration



There are several reasons why three virtual machines are needed. One of them is a configuration issue (e.g., if a migration is done from an old database version to a new one and the database interface does not allow the usage of both versions at the same time). Another reason is the migration between different Java versions. As RMI is compatible between different Java releases it provides a common representation of the migrated data. In many cases two virtual machines are sufficient, so VM1 and VM2 or VM1 and VM3 can be put together. However, at least two virtual machines are needed due to the nature of migration, ShapeShifter needs to work with two classes which share the same name at the same time. This is not possible in a single virtual machine.

The core of the migration process is done in the first virtual machine (VM1). The data from the original and the new object model are transported via RMI and transformed in a neutral representation. The migration is performed and the resulting data is saved in the new persistent object model.

As mentioned in the introduction, in its present implementation ShapeShifter provides an adapter for migrating object-oriented databases from the Versant Corporation. The Versant database is an object-oriented database, providing an interface that meets the requirements of the ODMG (Cattell, Berler, & Eastman, 2000) standard for accessing persistent object oriented data which is used by ShapeShifter. The present implementation of the Versant database provides support for Java Data Objects (JDO) as well, which is not used by ShapeShifter so far. JDO is the latest interface standard of the ODMG for transparent object persistence.

The interfaces of the components accessing the database are the "DB Manager," the "InstanceManager," and the "SchemaManager." Adapters for other databases can easily be implemented. Thus, ShapeShifter can be extended for use with Oracle or other well known databases. It is unnecessary to provide more information about the Versant database for explaining the concepts of object migration, as the user of ShapeShifter does not need to know details about the underlying database.

Basic Procedure of the Migration

The *Migration Engine* follows a fixed procedure during the database migration. The database as a consequence is always in a well defined state. Figure 4 illustrates the principal procedures of ShapeShifter during the migration of a database.

1. **Initialization of the environment.** Internal examinations and initializations are carried out. The different virtual machines are started and the required services are loaded and started.
2. **Parsing and validation of XMI input.** The *XMI Parser & Validator* reads the XMI input and examines it.
3. **Schema migration.** A new database is created with a new schema. The data from the old database is copied into the new one, as far as this is possible.
4. **Initialization with default values.** If specified in the XMI input, new attributes and classes will be allocated with default values.
5. **User-specific initialization.** The user-specific initialization program code, which is referenced in the XMI input, will be executed.
6. **Standard data migration.** The supported standard Data-Migration directives are executed as specified in the XMI description.
7. **User-specific data migration.** The user-specific data migration program code, which is referenced in the XMI input, will be executed.
8. **Examination of the new database.** The *Migration Verifier* examines the database, to assess whether or not it is in a consistent state and if it is accessible by the newly created schema.
9. **Cleanup environment.** The database and other components will be de-initialized. Memory will be de-allocated.

Accessing Persistent Objects During Migration

As shown in the previous section, ShapeShifter follows a determined workflow which always keeps the database in a well defined state. User-specific convertion or initialization code is executed at specific points within the scope of this workflow.

It must always be possible to access the old persistent object model as well as the new object model in a consistent state. This also applies when the migration is performed with distributed databases/systems. The component providing this functionality is the Instance Manager. It enables the user to access the original database as well as the new one at the same time. The original database only can be accessed read only, while the new database has read and write access.

At the beginning of the data migration, the new version of the database already contains all data from the original one, as far as the schema allows it. Accessing the database objects is done with the use of handles. Currently, only attributes of persistent objects can be accessed with ShapeShifter. Calling methods of persistent objects has not yet been implemented.

Figure 5. Accessing old and new objects during migration

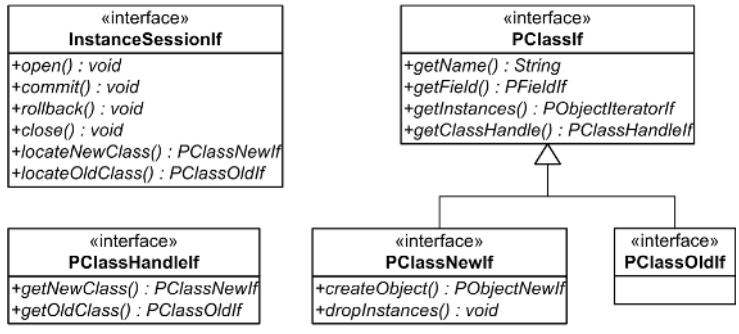
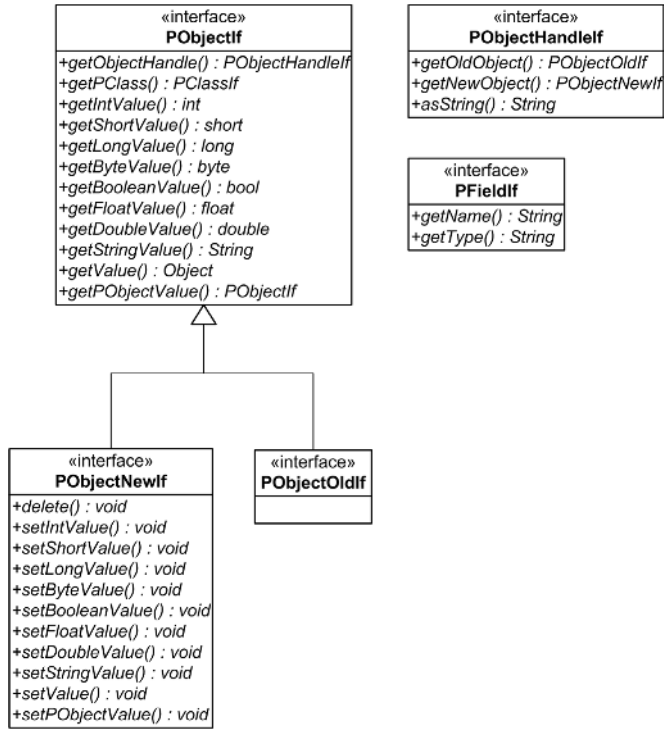


Figure 6. Accessing and manipulating instances of classes



To allow easy access to the database objects, ShapeShifter provides several interfaces for the user, as shown in Figures 5 and 6. The interfaces can be looked upon as providing two basic types of functionality: On the one hand, there are interfaces for finding classes and creating objects; on the other hand, there are interfaces for the manipulation of objects and attributes.

During the migration, the interface *InstanceSessionIf* provides the entry to these services. The methods *locateNewClass()* and *locateOldClass()* supply a handle of a persistent class. Thus, developers are able to access the original and the new classes or objects simultaneously. Of course, the new schema may differ from the original one in shape as well as in naming of classes and attributes.

The new and old classes can cross-access the old and new schema, respectively. In Figure 5, this is expressed through the generalization of *PClassNewIf* and *PClassOldIf* which both expand *PClassIf*.

With the method *getField()* in *PClassIf*, field descriptors for the access to the fields of the instances of the classes can be retrieved. The interface for *PfieldIf* is shown in Figure 6. Additionally *PclassIf* provides a *getInstances()* method to retrieve the object instances for the calling class. The new object instances can be manipulated in the ways permissible by the interface of *PObjectNewIf* in Figure 6. It could be noted that only object instances in the new database can be manipulated.

Example Migration

The following simple example demonstrates how a migration can be performed with ShapeShifter. For this purpose, the example makes use of the demonstration migration shown earlier in Figure 1.

Figure 7. XMI difference file for ShapeShifter

```
<XMI.difference xmi.id="_1" href="package|user">
  <XMI.replace xmi.id="_1.1" href="attribute|user.Company.address">
<XSHS.attribute xmi.id="_1.1.1" visibility="private" name="address"
                          href="class|user.Address">
     <XSHS.convert xmi.id="_1.1.1.1" type="String2PObjectNewIf"
                   href="class|converter.String2Address">
      <XSHS.source xmi.id="_1.1.1.1.1" href="attribute|user.Company.address"/>
      <XSHS.target xmi.id="_1.1.1.1.2" href="attribute|user.Company.address"/>
     </XSHS.convert>
   </XSHS.attribute>
  </XMI.replace>   <XMI.add xmi.id="_1.2" href="package|user">
   <XSHS.class xmi.id="_1.2.1" visibility="public" name="Address">
    <XSHS.superclass xmi.id="_1.2.1.1" href="class|java.lang.Object"/>
    <XSHS.attribute xmi.id="_1.2.1.2" visibility="private" name="zip"
                          href="basetype|String"/>
    <XSHS.attribute xmi.id="_1.2.1.3" visibility="private" name="street"
                          href="basetype|String"/>
    <XSHS.attribute xmi.id="_1.2.1.4" visibility="private" name="city"
                          href="basetype|String"/>
   </XSHS.class>
  </XMI.add>
</XMI.difference>
```

At first, the class *Company* contains a simple s*tring* attribute *address*. Next, an evolution in the datamodel takes place and a new class called *Address* is created that is referenced from the class *Company*. The data from the original *address* strings must not be deleted and has to be inserted in the appropriate fields in the new *Address* class.

To accomplish this task, two steps have to be performed: First, a converter class needs to be created, which copies the original data from the attribute *address* of the class *Company* into the attributes *street*, *city,* and *zip* of the new class *Address*. Second, an XMI difference file has to be written, which calls the converter class and creates the new class Address with the appropriate attributes.

The XMI difference file in Figure 7 describes the exact procedure of the migration for ShapeShifter. Compared to the XMI difference file in Figure 1, it is interesting to see that the file has been extended with a few *"XSHS-Tags."* These extensions to the XMI-Standard are necessary for ShapeShifter to perform the migration properly. The extensions consist of directives to include appropriate plug-ins, respectively, user-specific converters.

The XMI file is divided into two parts: the *XMI.replace* part and the *XMI.add* part. The *XMI.replace* part converts the original data from the *address* string to the corresponding fields in the new *Address* class. This is realized by the converter class *String2Address* using the plug-in framework of ShapeShifter. The following tag in the XMI file is the *XSHS.attribute* tag, which may seem redundant, but is actually needed if the visibility of the attribute has to be changed. The next tag, the *XSHS.convert* tag, specifies the converter class to be used, with the source and target of the conversion specified by the *XSHS.Source* and the *XSHS.Target* tags, respectively.

In the *XMI.add* part, a new class called *Address* with the corresponding attributes is created which is self-explanatory when looking at the XMI file.

Figure 8 shows the user-specific converter class needed for this migration. As mentioned before, the execution of this class is specified in the XMI file. The task of the converter

Figure 8. Converter plug-in

```
public class String2Address implements CString2PObjectNewIfIf
{
  private static PFieldIf pfStreet, pfCity, pfZip;
  private static PClassNewIf pclass = null;
  public PObjectNewIf convert(String source) {
    PObjectNewIf result = null;
    // The follwing variables have to be set only once
    if (pclass == null) {
      InstanceSessionIf session = InstanceSessionManager.getCurrent();
      pclass = session.locateNewClass("user.Address");
      pfStreet = pclass.getField("street", "java.lang.String");
      pfZip = pclass.getField("zip", "java.lang.String");
      pfCity = pclass.getField("city", "java.lang.String");
    }
    // Create the Adress Object and set the variables
    result = pclass.createObject();
    result.setStringValue(pfStreet, extractStreet(source));
    result.setStringValue(pfCity, extractCity(source));
    result.setStringValue(pfZip, extractZip(source));
    return result;
  }
}
```

is to decompose the *address* string from the old class and to insert it into the appropriate fields in the new class.

When called, the converter is delivered the address string from the source class by the migration engine. If the converter is executed for the first time (*pclass==null*) it has to acquire the references for the attributes, but this has to be done only once.

With the method *pclass.createObject()*, a new address object is created. The attributes are set with the method *setStringValue()*. The methods *extractStreet()*, *extractCity()* and *extractZip()* are used for the decomposition of the appropriate data from the *address* string. The implementations of the extracting methods are not shown here as they are negligible.

The combination of the XMI file from Figure 7 and the converter plug-in from Figure 8 serves as input for ShapeShifter. The migration of the schema and the data can then be performed automatically.

Conclusion

Migration of object models is a common task in today's agile software development, particularly in case of code refactoring. Suitable tools for the corresponding data migration are not yet available and much work must be done by hand. The tools provided by the database vendors can only handle small parts of the work and fail when complex migrations have to be performed (Nierstrasz & Tsichritzis, 1995).

In this domain, ShapeShifter provides a solution for the object oriented database Versant (n.d.). However, the market share of the Versant Corporation is very small compared to IBM or Oracle. Hence, the next step in the development of ShapeShifter should concentrate on the adaption of other database products. Other concepts are imaginable as well, like migration between different database vendors, such as Versant and ObjectStore, or even between different database technologies, such as relational and object-oriented databases.

Another area of application could be software testing. In automatic regression testing environments like JXUnit (n.d.) with separated test data, schma changes of the application not only entail the migration of the application data but also the migration of the complete set of test data. ShapeShifter can be used to migrate the test data automatically to the new schema and hence perform the migration of test data more efficiently.

References

Ambler, S. (2002). *Agile modeling: Effective practices for extreme programming and the unified process*. New York: John Wiley & Sons.

Beck, K. (2000). *Extreme programming explained: Embrace change*. Boston: Addison Wesley.

Cattell R., Berler M., & Eastman J. (2000). *The object data standard: ODMG 3.0*. San Fransisco: Morgan Kaufmann.

Eclipse Foundation Internet Homepage. (n.d.). Retrieved July 10, 2004, from *http://www.eclipse.org*

Fowler, M., & Kendall, S. (2003). *UML distilled second edition: A brief guide to the standard object modeling language*. Boston: Addison-Wesley.

Grose, T. (2002). *Mastering XMI: Java Programming with XMI, XML, and UML*. Indianapolis, IN: Wiley

Harold, E., & Means W. (2002). *XML in a nutshell second edition: A desktop quick reference*. Sebastopol, CA: O'Reilly & Associates.

IBM Alphaworks Internet Homepage. (n.d.). Retrieved July 10, 2004, from *http://alphaworks.ibm.com*

JXUnit Internet Homepage. (n.d.). Retrieved July 10, 2004, from *http://sourceforge.net/projects/jxunit/*

Nierstrasz, O., & Tsichritzis D. (1995). *Object-oriented software composition.* Indianapolis, IN: Prentice Hall.

Versant Internet Homepage. (n.d.). Retrieved at July 10, 2004, from *http://www.versant.com*

Waller, E. (1991). Schema updates and consistency in DOOD'91. *Proceedings: Lecture Notes in Computers Sience* (Vol. 566). Berlin: Springer-Verlag.

Chapter VI

PRAISE: A Software Development Environment to Support Software Evolution

William C. Chu, Tunghai University, Taiwan

Chih-Hung Chang, Hsiuping Institute of Technology, Taiwan

Chih-Wei Lu, Hsiuping Institute of Technology, Taiwan

YI-Chun Peng, Tunghai University, Taiwan

Don-Lin Yang, Feng Chia University, Taiwan

Abstract

Responding to the fact that software systems become more and more complex and mutable, not only the software-standards-related technologies should be adopted, but the environments for software development and evolution should also be flexible and integratable. These facts make software development and maintenance difficult and costly. In this chapter, we first illustrate the activities and studies for software standards, processes, CASE toolsets, and environments. Then, we propose a process and an environment for evolution-oriented software development, called the PRocess and Agent-based Integrated Software development Environment (PRAISE). PRAISE advocates software development with popular software methodologies, and it uses an XML-based mechanism to unify the various paradigms with different standards. It integrates processes, roles, toolsets, and work products to make software development

more efficient. With PRAISE, users are encouraged to adopt familiar mechanisms and formal approaches as they wish. PRAISE maintains the consistency of the paradigms so that users do not need to worry about conflicts with other paradigms that are built in or added later. PRAISE meets the need for evolving software development and maintenance.

Introduction

Software must keep evolving to meet the variability of the software requirements, system environment, users, and so forth. As modern software expands, it also becomes increasingly complex, making software development more difficult than before. This complexity naturally affects software evolution. Changes to software can come up at any time in the software life cycle. If developers do not consider the demands of software evolution during the primary stage of development, later changes can be very costly.

In software evolution, the first challenge is design recovery. Maintainers need to do more because documents often become inconsistent with the system. Secondly, the architectures of some software systems are poor or lack flexibility. Maintainers may get half the results with twice the effort due to the built-in drawback of software. They cannot improve the quality of the software easily since these are architecture-level problems. Thirdly, the quality of software heavily depends on the standard methodologies, supporting tools, process management, developer expertise, domain knowledge, and the extent of the integration of the factors. Currently, most of the activities for software development are quite ad hoc. Standardization is rarely applied, and changes are usually implemented manually. Software development involves many activities in many phases with different types of stakeholders who play different roles and produce artifacts in a collaborated way. Without a proper modeling approach and tool support, the performance of these activities will be poor, and the handling of artifacts produced from these activities will be error-prone and sometimes too overwhelming.

Thus, software systems nowadays face more challenges. One of the toughest challenges is teamwork development and integration. Therefore, standardization becomes vital and important for effective software development and maintenance. Many software standards, such as Unifying Modeling Language (UML), design patterns (DPs), and commonly accepted standard mechanisms, such as component-based approaches, have been proposed and advocated to improve software productivity and reduce the high cost of software.

Current standard methods and mechanisms usually only cover part of the software process. For example, UML provides standardized notation for modeling software analysis and design, yet lacks support to the implementation and maintenance phases. DPs help developers in the design phase, while component-based technologies focus on the implementation phase. Because these standards do not talk to each other, designers need to spend much manual effort to map and integrate these standards. Without unifying and integrating these standards, reused parts of the software process will still be very less, so that the development cost will become more.

In our previous studies (Chu et al., 2002; Lu, Chu, Chang, Lian, & Yang, 2003), we proposed an XML-based unified model, called XUM, which can integrate and unify a set of well-accepted standards of a system into a unified model represented in XML. We have demonstrated the feasibility, with XUM, to overcome the inconsistency problem of software artifacts inherent in ripple effects during software development and maintenance. However, it still lacks many important features when applied to a real case of software development, such as process modeling, role modeling, the mechanism of integrating with CASE tools, and so forth. In this chapter, we propose an integrated software development environment called PRocess and Agent-based Integrated Software development Environment (PRAISE), which can be tailored to support software development and evolution much more effectively.

This chapter is organized as follows. First, the problems and related studies of software development environment are discussed. The next section illustrates the details of PRAISE. And the last section offers a conclusion and discusses future works.

Software Development

The Software Development Environment and Process

There are numerous factors such as organization scale, applied software process, and specific projects which may significantly affect the software development process. Moreover, software methodologies, technologies, supporting tools, and process managements also vary frequently in order to accommodate specific requirements. A supportive software development environment can help the designer to solve some problems of software development in a cost-effective way. Software development environment (SDE) is a comprehensive, highly integrated set of tools supporting the complete software development process (Engels, Lewerentz, Nagl, Schäfer, & Schürr, 1992). SDE has been designed to effectively support software development (Harrison, Ossher, & Tarr, 2000). However, most existing SDEs still lack some important features necessary to be able to support software development more effectively. For example, the programming support environment only focuses on the assistance of coding and does not cover other phases of the software engineering process (Habermann & Notkin, 1986; Workflow Management Coalition (1994). Software engineering environments (Ossher, & Harrison, 1990; Wassermen, Pricher, Shewmake, & Kersten, 1986) integrate a collection of tools that facilitate software engineering activities. Process-centered software engineering environments (Finkelstein, Kramer, & Nusibeh, 1994) have provided a powerful means of integrating processes and tools, and partially automating tasks. However, some problems still cannot be solved.

The major problem of software development is that elements of a software document are not integrated, and the information used to combine and integrate the shared/general semantics must be remedied by human labour. One way to stem this problem is to represent and integrate software document elements into a common form. Mi and Scacchi (1996) provided a metamodel for formulating knowledge-based models as a Unified

Resource Model (URM), which integrates characteristics of major types of objects appearing in software development models. URM also includes specialized models for software systems, documents, agents, tools, and development processes. URM serves as the basis for integrating and interoperating a number of process-centered CASE environments. Although URM provided a very good general conceptual model and tool integration mechanism, it was not concerned with the issues of applying it to support software development and evolution in an integrated environment.

The task of software development cannot be achieved simply by only modeling techniques, but it needs effective cooperation from all of the tools/models through the software process. The message and information exchange among these cooperative models becomes another key problem of the process. MOF and XMI provided a possible solution to this problem.

The Meta Object Facility (MOF) (Object Management Group, 2002) is the foundational technology for describing object models, which cover the wide range of object domains: analysis (UML), software (Java, C++), components (EJB, IDL, CORBA Component Model), and databases (CWM). In addition, the XML Metadata Interchange (XMI) specification (Object Management Group, 2003) defines technology mappings from MOF metamodels to XML DTDs and XML documents. These mappings can be used to define an interchange format for metadata conforming to a given MOF metamodel. XMI is a widely used interchange format for sharing objects using XML. Sharing objects in XML is a comprehensive solution that builds on sharing data with XML. In our approach, we use MOF and XMI standards as an interface to communicate and cooperate with other CASE tools.

The XML-Based Unified Model (XUM) (Chu et al., 2002; Lu et al., 2003) is a representation of artifacts of software systems defined in XUMM (XUM Metamodel). These artifacts (i.e., software products) are the standardized modeling information collected from submodels of paradigms used in each phase of the software life cycle; they are integrated into an XUM document of a system with XUMM, by revealing the interrelationships of the artifacts. However, XUM only integrates the artifacts of the software process; it does not deal with processes themselves, along with toolsets and roles. In the next section, we will extend XUM to cover definitions of processes, participants, artifacts, and applied tools of the software development.

Maes and Wooldridge (1997) provided important definitions of the agent. An agent was defined as "a computational system which is long-lived, has goals, sensors, and effectors, and decides autonomously which actions to take in the current situation to maximize progress toward its [time-varying] goals," specifying that the software agent was a particular type of the agent, inhabiting computers and networks, and assisting users with computer-based tasks (Maes, 1997). We also have adopted agent paradigm to implement the coordination of different models in PRAISE.

Related Work

PRAISE has adopted some of features and ideas from the works of PRIME (Lu et al., 2003), RUP (Pohl et al., 1999), and URM (Maes, 1997). PRIME is a framework for PRocess-

Integrated Modeling Environments. PRIME has integrated tool and process models together. It also adapts agents to help the software developing process. PRIME is an implemented framework for process-integrated tools and process engines.

RUP is a software engineering process framework used to enhance team productivity and to deliver software cases via guidelines, templates, and tool guidance for all software development activities. RUP is an implementation of process-integrated environment (PIE). In most cases, RUP is general and complete enough; it can be modified, adjusted, extended, and tailored to accommodate specific characteristics, constraints, and capabilities of the software development for required business rules. While supporting a specific case, RUP recombines predefined activities to configure an appropriate software process, using a process engineering toolkit. The representation of a tailored process is a set of HTML pages, which can be viewed using Web-based browsers. RUP can serve as a knowledge base for software developments. Nevertheless, since the processes/performances and the performances' guidance are not integrated properly, the engineers have to interact with the "passive" separated guidance tools frequently; that is, users need to perform the development tasks and feedback the status of the task performances, either the states of work products or the performed actions. Hence, the guidance offered by RUP is still limited.

Unified Resource Model (URM) is a knowledge-based metamodel which integrates characteristics of major types of objects appearing in software development models. URM includes specialized models for software systems, documents, agents, tools, and development processes. URM has implemented a prototype, which serves as the basis for integrating and interoperating a number of process-centered CASE environments.

PRAISE is an XML-based metamodel for a process and agent-based integrated software development environment. PRAISE includes both the external representation in UML and its internal representation in XML, and can be used to support the integration of software development in a global aspect.

We summarize the major characteristics of these software development environments in Table 1.

From the discussions and Table 1, it is clear that PRAISE has many beneficial characteristics and aims to provide more practical features to the environment for effective software development and evolution.

PRAISE: An Environment for Evolution-Oriented Software Development

The primary objectives of PRAISE are to support software development and evolution effectively. The major features are summarized as follows:

Table 1. Major characteristics of four software development environments

	Process Integration	Data Integration	Agent-based development	Process Management	Customize Process	Visual	Tool Integration	Document Driven	Process Driven	Workflow Guidance	Variation Control	Guidance	Role Modeling	Document Integration	Ripple Effect Analysis
PRAISE	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V
PRIME	V	V	V	V	V	V	V		V	V	V	V	O		O
RUP	V	V		V	V	V			V	V	V	V			
URM	V	V	V	V	V	V	V	V	V				V	V	

Legend: V - have this feature; O - partially

1. Design a software process engineering metamodel which can be customized or tailored to describe a concrete software development process to meet the specific project's needs.
2. Implement a workflow engine that realizes the tailored software development process based on the semantic of the software process engineering metamodel, and then take over the actual enactment process (i.e., planning and executing a project using the tailored software process described with metamodel).
3. The software development process is performed by collaborating actual development activities, such as "Understand Stakeholder Needs," "Defined System Architecture," and so on. The collaboration process is controlled and facilitated by the workflow engine according to the definition of tailored software development process. Each of the performed roles manipulates separate Computer-Aided Software Engineering (CASE) tools with particular functionalities, such as requirement tools, design and analysis tools, testing tools, and so on, or to be assisted by other alternative guidance, such as checklists, templates, guidelines, and so on.
4. Make the software production process achieve better process quality and product quality at decreased costs in order to promote the competition ability of software industries.

The Architecture of PRAISE

PRAISE is composed of three domains: the methodology modeling domain for the process build time, the process enactment domain, and the process performance domain for the process run-time, as shown in Figure 1.

Methodology Modeling Domain (MMD)

Process is the key to good software development practices. For this reason, in the methodology modeling domain shown in Figure 2, the necessary modeling elements should be well-defined, which then can provide the software process engineer rich semantics to describe the variant software development process. PRAISE adopts a well-accepted process model, the Software Process Engineering Metamodel (SPEM) developed by OMG (2003) to provide a standard metamodel which emphasizes the software process engineering. However, some modeling definitions about performed roles, work products, and tools are still very poor. The detail of the model about MMD will be described later.

Process Enactment Domain (PED)

The process enactment domain (see Figure 3) is responsible for the process enactment which is implemented by managing the control logic of coordination between separate tools. Similar to the definition in WFMC's Workflow Reference Model (Workflow Management Coalition, 1994), the process enactment service is responsible for instantiating process instances utilizing the workflow engine, which is in charge of interpreting the process definition, to enact process instances according to the interpretation in order to coordinate distinct responsible performers' contributions to achieve project's goal. Two logical separated control mechanisms constitute the process enactment service: the workflow engine and the agent-based interactive mechanism.

The Workflow Engine

The workflow engine is the core component in this domain is engine, which maintains the running process instances according to the interpretations of the process definition. It handles internal control data and relevant work products for determining flow-control logic and interacts with CASE tools in the PPD, as shown in Figure 4, via Enactment Service Interface. Enactment Service Interface is an invoked application interface which enables the workflow engine to indirectly activate a specific tool for responsible performer to execute a particular activity, and vice versa.

The Agent-Based Interaction Mechanism

The socket is a common technology for implementing a particular protocol, but in PRAISE, we implement it by the intelligent agent mechanism (see Figure 5) because of its two characteristics: intelligence and mobility. Two sorts of agents utilize the respective characteristics.

PEn Agent is implemented as an intelligent agent. As the descriptions of the PED, the enacted process is determined according to the various conditions of state transitions.

Figure 1. PRAISE architecture

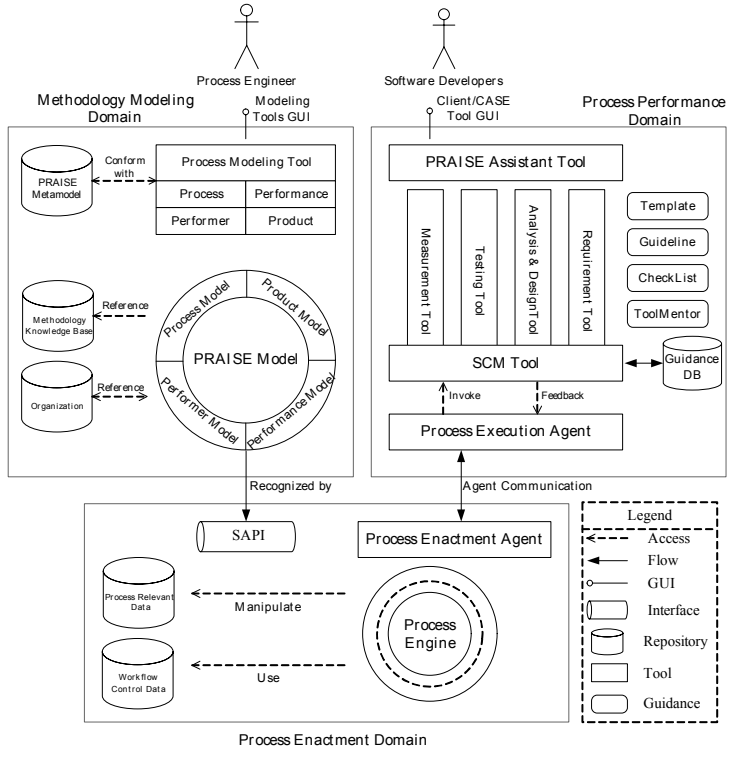


Figure 2. Methodology modeling domain

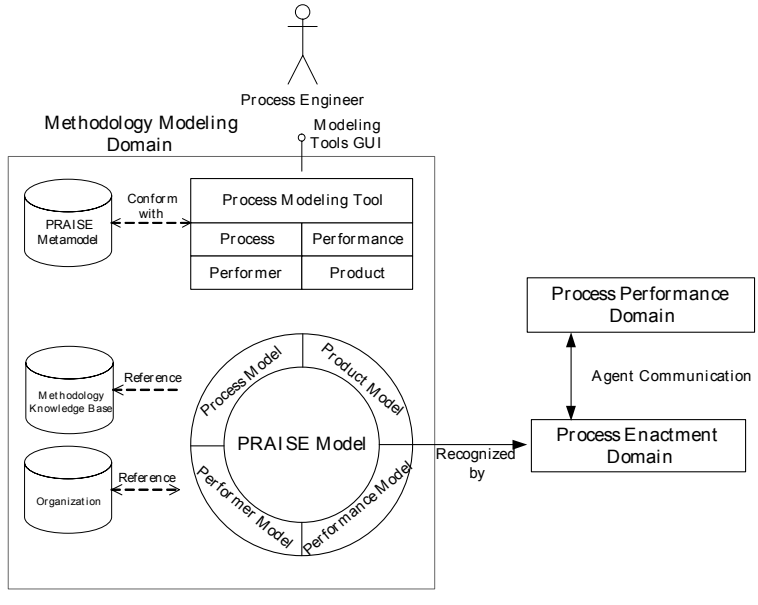
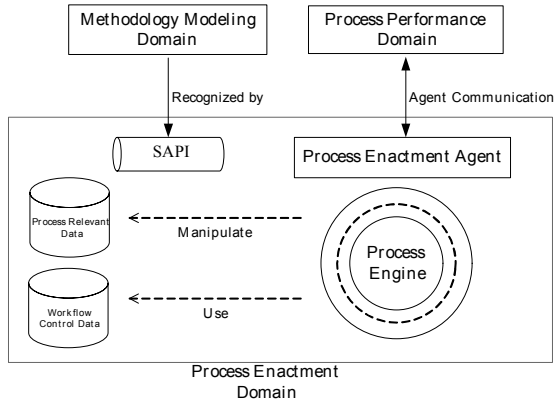
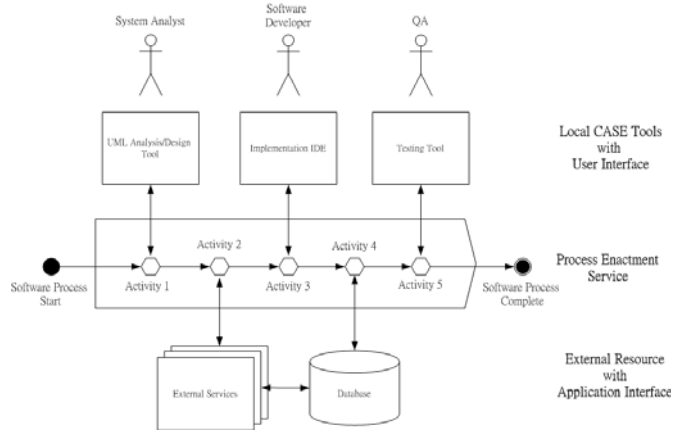


Figure 3. Process enactment domain



Based on numerous factors, such as the product states or feedback data, those conditions can be formulated as several rule clauses and fact with formal format that may be Simple HTML/XML Ontology Extension (SHOE), XML-based Ontology exchange Language (XOL), Resource Description Framework (RDF), and so on. Such ontology groups the control-logic knowledge base that is imposed to the PEn Agent. During the process, the agent may infer which activities should be executed next from the imposed knowledge base and feedback data, and then pass corresponding directives to the PEx Agents to execute next activities. The PEn Agent iteratively directs PEx Agents until the finish conditions of the software process are all satisfied. Briefly, the collaborations of PEn/PEx Agent pairs are namely the familiar master/slaves paradigm. As a result, PEn agents could be the delegations of flow engine and the traditional workflow management will be displaced from centralized to distributed, and computational loads of the workflow engine could be reduced substantially but increase the flexibility and scalability for implementation purposes.

Figure 4. Coordination between distinct process performer, supporting tools, and activities



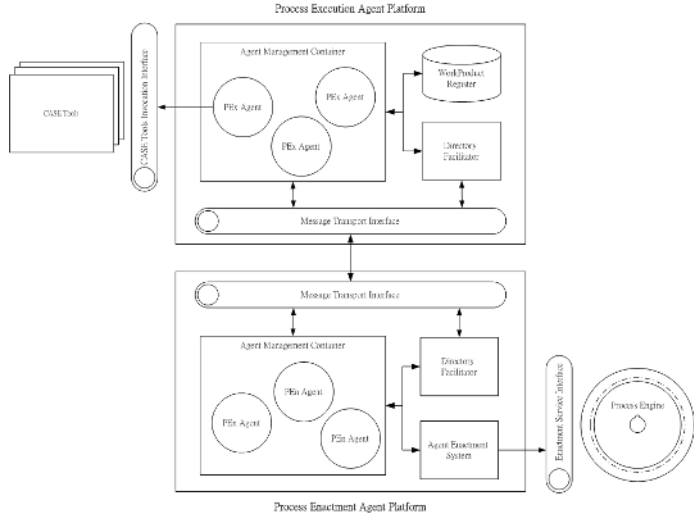
PEx Agent is implemented as a mobile agent. A software development is achieved by several activities' executions in particular order. The traditional transitions mechanism was implemented primarily using the relational database, but all the necessary data placed into the database will cause frequent access communications while numerous activities are running concurrently. The motilities of agents may well be an alternative technology to implement the transitions mechanism; for instance, a mobile agent may carry the output products away from the finished activity execution, and bring them as an input parameter to the next execution in other host, to simplify the implementation of the process enactment system. It may make the transitive behaviors closer to the essential of the workflow paradigm in nature.

The agent-based interaction mechanism provides message exchange facilities which are required for the interaction between the PED and PPD; for instance, a tool request from the PED has to be directed to the PPD in order to invoke the tool responsible for performing the requested activity according to the process definition. On the other hand, the invoked tool also needs to feed back the relevant product created in the activity to the PED and then keep on furthering the process.

Process Enactment Agent Platform (PEn-AP)

For internal, the PEn-AP deployed in the PED consumes enactment directed from the workflow engine and communicates with the process performance agent platform deployed in the PPD for external. Applying to the FIPA's Agent Management Reference Model (Foundation for Intelligent Physical Agents, 2000), the process enactment agent platform provides the physical infrastructure in which process enactment agents (PEn-Agent) can be deployed. The PE-AP is formed by the agent management container, numerous PEn-Agents, directory facilitator, agent enactment system, and message transport interface.

Figure 5. Agent-based interaction mechanism



The agent management container provides a run-time execution environment for agent instances and controls their life cycles, conceptually, the agent's life cycle related with the life cycle of the process instance.

The directory facilitator provides agents the directory services to facilitate the agent-query mechanism.

The agent enactment system receives the enactment form the workflow engine and is responsible for instantiating the corresponding PEn-Agent with given rule clauses which are produced by interpreting the process definition. After the instantiation, the agent will be registered on the directory facilitator and the container will take over its life cycle.

The message transport interface (MTI) provides a message-oriented communication mechanism between the process enactment and execution agent platforms. PEn Agents encapsulate the directive to a message and pass to PEx Agents through MTI and vice versa. In addition, the requests of DF services also are passed by MTI.

Process Execution Agent Platform (PEx-AP)

PEx-AP provides an infrastructure and run-time environment in which PEx agnets are deployed and responsible for activities transitions. Most structural building blocks of PEx-AP are similar with PEn-AP except for the *WorkProduct Register*. During the process execution, there are several products that cannot only be used in an activity but server as global parameters shared by many activities in order to complete necessary works. This kind of product can be registered into the work product register and share the product instance with other PEx agents.

Process Performance Domain (PPD)

The process performance domain shown in Figure 6 provides a run-time environment, within which allows assigned tools to be invoked by PEx agents that delegate the enactment service in PED through the tools invocation interface according to the process definition to support the particular activities' executions. By interacting with the generic PRAISE client tool, the development team makes hand-on use of distinct CASE tools along with a general assistant tool, which is responsible for presenting development guidance to support activities' execution in different ways (e.g., templates, guidelines, tool mentor, or checklist) to improve the process and product qualities while some sort of manual and creative activities are carrying out, such as "Analyze the Problem" or "Understand Stakeholder Needs," Activities may not be easily fully supported with the requirement tools. The PEx agents may just directly access external resources without invoking CASE tools deployed in the PPD once the executing activity is set as automatic. An external resource may be a stand-alone distributed service, Web services, or a database.

The PRAISE Model

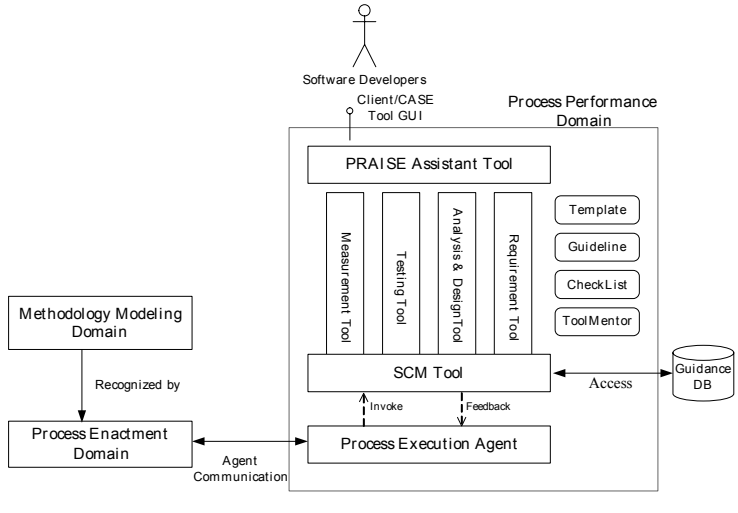
In PRAISE, the software developer or maintainer may perform the tasks to produce artifacts in software phases — analysis, design, coding, testing, and maintenance — with software standards like UML. PRAISE offers users guidance by revealing the sharing and integrating information of these artifacts. PRAISE also provides service to manage individual role's tasks and interactions of the participants of the project, so as to foster team cooperation. All the information manipulated with PRAISE is modeled and represented in an XML-based document, called a PRAISE model.

Extending the idea of the Software Process Engineering Metamodel (SPEM) (Object Management Group, 2003), we have designed 1 + 4 submodels to support the PRAISE model (see Figure 7). The 1 + 4 model is one foundation model plus four submodels: process model, performer model, performance model, and product model.

- **Foundation Model:** The *foundation model* provides the primitive semantic for the other four submodels. The elements of the foundation model are described as follows.

ModelElement is the primitive component of all models in PRAISE. *Classifier* is an element that describes behavioral and structural features. It comes in several specific forms, including class, data type, interface, component, artifact, and so forth. *Operation* defines the dynamic behaviors of a class. *Parameter* is a variable that can be changed, passed, or returned by the operation, classifier, or model elements. *Generalization* is the relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional

Figure 6. Process performance domain



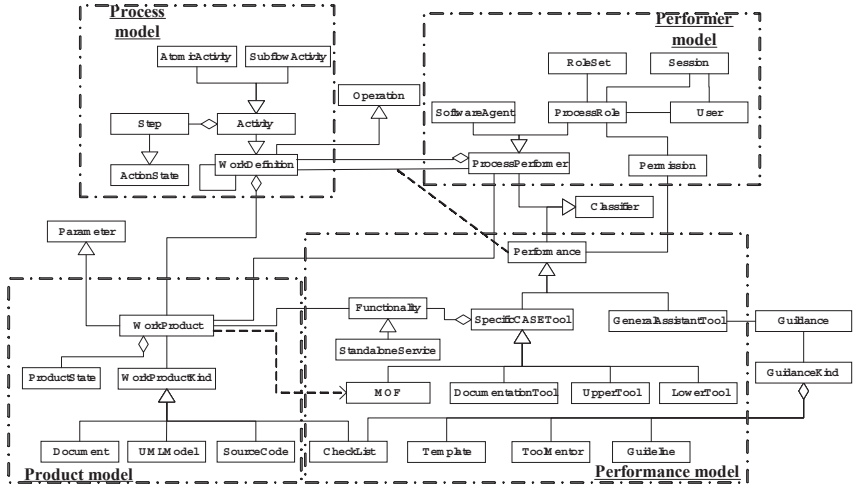
information. *Association* is the relationship between two or more classifiers that specifies connections among their instances. *Dependency* is the relationship between process model elements in which a change to one modeling element will affect the other modeling element. *Composition* is a relationship, which represented an element, is a composite of other elements. *Guidance* elements can be utilized to provide the performer some useful knowledge for the associated model elements. Guidance is a Model Element associated with the major process definition elements, which contains additional descriptions such as techniques, guidelines and procedures, standards, templates of work products, examples of work products, definitions, and so on. The structure of the foundation model is shown in Figure 8.

- Process Model

  Process Model defines the phases of a software process that is adopted in a specific software development. It offers users options for generic and tailored design processes, which suit the needs of software diversity. Figure 9 illustrates the structure of the process model.

The process model defines the structural model elements that are utilized to construct a software development process by the process engineer. *WorkDefinition* is a model element of a process. A work definition describes what a process role performs. *Activities* are the main element of work. There are two elements that are specialized from the Activity element: AtomicActivity and SubflowActivity. *AtomicActivity* is composed of several *Step* elements in sequence. Each step stands for the preceding activity's states. *SubflowActivity* element presents a subprocess, which includes certain non-atomic activities.

Figure 7. The structure of the PRAISE model



- **Performer Model:** *Performer Model* defines the attributes of roles and their relations with other model paradigms. The roles in the software development process include designer, users, and software agents, which involve or enact the software development. Figure 10 shows the structure of the performer model.

The performer model defines those elements that stand for the performers in the software development process. *ProcessPerformer* is responsible for work products and performing assigned activities. ProcessPerformer may be defined as a *ProcessRole* or a *SoftwareAgent*. ProcessRole is a real developer in the development process.

The access control mechanism is also an important subject to the performer model. Each *User* is assigned one or more process roles, and each process role is assigned one or more permissions. A process user establishes a *Session* in the run time, during which the user acts as some roles that they are a member of, thereby acquiring the roles' permissions. *RoleSet* organizes those roles with closely related capabilities involved in the software process.

- **Product Model:** *Product Model* defines and specifies the contents of all software products in the software development process. It is used to define product information of analysis, design, source coding, design guidance, etc. Figure 11 is the structure of the product model.

The product model defines the elements which may be produced, consumed, or modified by performers described in the performer model. *WorkProduct* can be treated as the parameters of a work definition. *ProductState* defines the possible states in a work product's life cycle. *WorkProductKind* describes a particular category of work products. From the parameter's perspective, it means the data type of work products. There are four possible types defined in the PRAISE model; they are Document, UMLModel, SourceCode, and CheckList.

Figure 8. The structure of the foundation model

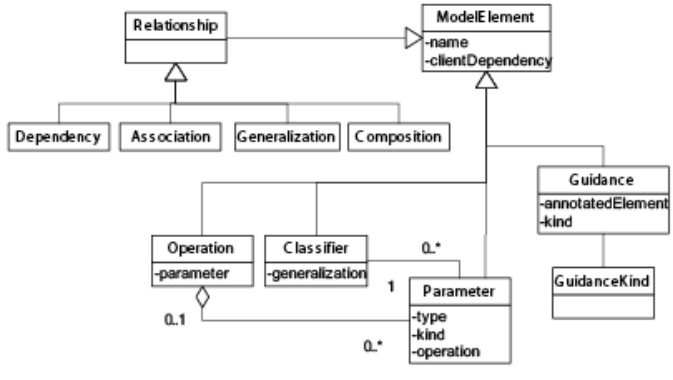


Figure 9. The structure of the process model

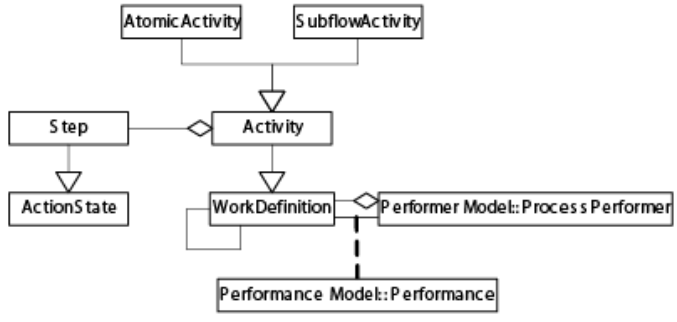


Figure 10. The structure of performer model

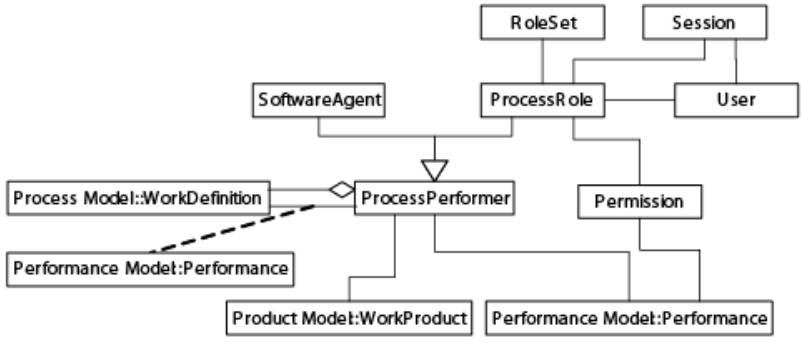
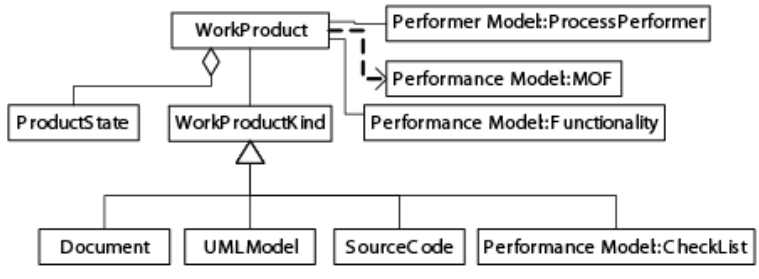


Figure 11. The structure of the product model



1. *Document* simply represents any type of text documents (e.g., Glossary, Risk List, Software Development Plan, hardware specification, etc.).
2. *UMLModel* represents those UML models that may be involved in the process, including the use case model, analysis model, design model, and implementation model.
3. *SourceCode* represents all the source code written by responsible programmers in particular programming languages, such as C++, Java, and so forth.
4. *CheckList*, which was described as a guidance type in the foundation model, also represents a special kind of parameter in the product model.

- **Performance Model:** *Performance Model* defines the information that is needed by the toolset of PRAISE to assist users to develop or maintain software. Figure 12 is the structure of the performance model.

The performance model is defined as the actual performances of work definitions which are defined in the process model. The actual performances include a general assistant tool and specific CASE tools.

The main class *Performance* in this model is actually an association class that is used to provide more semantic meanings on the connection between a work definition and its responsible role. Performance includes three elements: *SpecificCASEtool*, *AssistantTool*, and *MOF*.

SpecificCASEtool describes specific capabilities of a CASE tool. Those tools are categorized into three kinds according to a different purpose in each development phase, and they support the corresponding work product kinds. *Functionality* is modeled on the capabilities of tool. *StandaloneService* describes software services, like RMI, CORBA, and so on that may be deployed on standalone hosts. *GeneralAssistantTool* describes the tools, that support the manipulations of CASE tools for performers with some useful guidance.

Guidance displays the information that assists developers to perform those tools. There are four guidance kinds: *ToolMentor*, *Guideline*, *Template*, and *CheckList*. *ToolMentor* shows how to manipulate corresponding tools. *Guideline* describes the practical information, techniques, and advice about how to perform tasks or work products. *Template* is ready-to-use or semi-finished documents with "standard" formats. The template can be quickly applied to document-based work products for standardized intention. *CheckList* is a special element that refers to guidance, and work products. A checklist may be a kind of guide for performers. A checklist also can be authorized and modified by the process engineers using the modeling tool.

The *MOF* elements describe the information in XMI format which will be exchanged with other tools, including CASE tools.

- **The Collaboration of Four Models:** The relationships of the 1+4 models are shown in Figure 13. The foundation model provides the backbone semantics and basic elements for the other four submodels. The process model defines the structural

Figure 12. The structure of the performance model

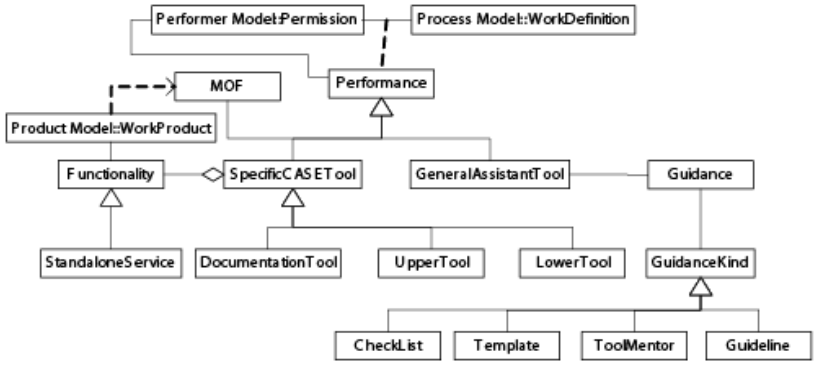
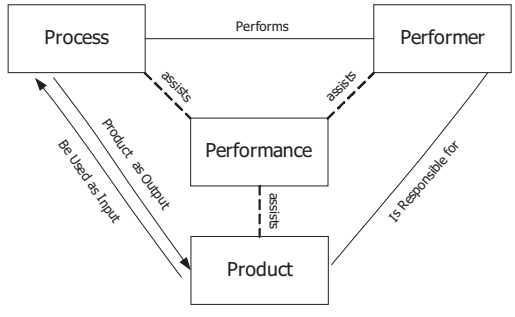


Figure 13. The interaction relationship between the four models



model elements that are utilized to construct the software development process by the process engineer. The product model defines the data model elements that are work products which may be produced, consumed, or modified in the software development process. The performance model tries to identify the actual performances of work definitions in the tool-based approach.

An integrated PRAISE model is formed by interrelating the 1+4 submodels. Primary interrelations are summarized as follows:

1. ProcessPerformer element connects WorkProduct element with "is responsible for" association relationship.
2. WorkDefinition element relates WorkProduct element with the aggregation relationship.
3. ProcessPerformer element connects WorkDefinition element with "perform" association relationship that is further represented by Performance association class.

4. All model elements and relationships in PRAISE model are extended from general elements in the foundation model.

The PRAISE model is used to support the software evolution process, which is performed by several roles. Each role is responsible for certain work products, which are produced by the development activities.

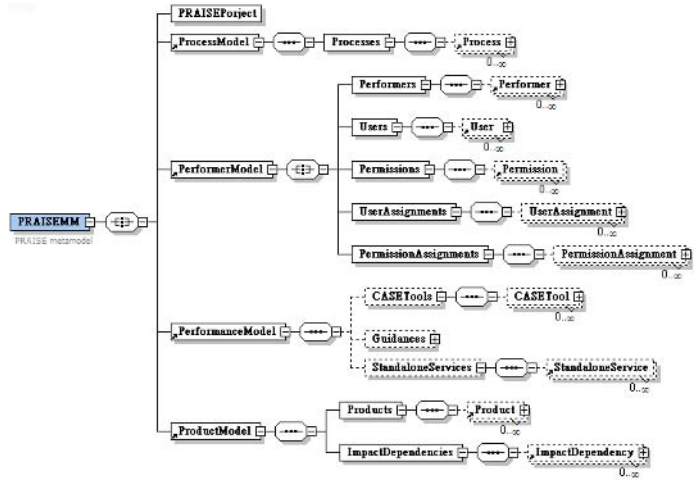
These work products are utilized as the output of finished activities; and they will be the input of continuing activities. Further, the activities they perform will be supported by specific CASE tools or alternatives, fully or partially.

PRAISE provides a generalized model that can be easily tailored to specific environments for different projects. We define the PRAISE MetaModel, *PRAISEMM*, which is specified with XML schema, defines the structure of a PRAISE model. The relationship of the PRAISEMM with a PRAISE model is similar to that of the DTD with an XML document.

The Metamodel of PRAISE

In PRAISEMM, we define three primitive elements: *Component*, *Relation*, and *Integration_link*. *Component* describes the ingredient information of models. *Relation* represents the relationships/associations among components in models. *Integration_link* is used to link/connect a set of components or relations that have the same semantics but may be named or represented differently in different paradigms. Integration_links, which are implemented with xlink (Thompson, 2003) and IDREF(s) of XML technology, are one of the key features for the modeling information integration. Through these underlying interconnections, the paradigms, which adopt various standards that might

Figure 14. The structure of PRAISEMM



share some semantics but which were not explicitly represented, can be integrated and unified in a PRAISE model.

Figure 14 shows the structure of the PRAISEMM, and Figure 15 shows partial semantic of process model and representation in a well-accepted language, XML.

The Toolset of PRAISE

PRAISE consists of three domains: methodology modeling domain, process enactment domain, and process performance domain. In PRAISE, three corresponding executable components respectively support the three domains:

1. **PRAISE Definition Tool** is a Java-written stand-alone application that supports the methodology modeling domain.
2. **PRAISE Enactment Service** is a Java-written demand service that supports the process enactment domain.
3. **PRAISE Client Tool** is a Java-written stand-alone application that supports the process performance domain.

Before detailing the functionality of a provided tool, Table 2 shows the convention of icons that are used in PRAISE. Note that we have used notations similar to RUP, so the learning curve from the user of RUP will be much lower.

Table 2. The icon notation of PRAISE

	Element	Icon		Element	Icon
Process	AtomicActivity		Work Product	Document	
	SubflowActivity			UMLModel	
Performer	ProcessRole			SourceCode	
	SoftwareAgent		Guidance	CheckList	
Performance	Documentation tool			Template	
	Upper tool			Guideline	
	Lower tool			ToolMentor	

Figure 15. Partial XML schema of the process model

```
<xs:element name="ProcessModel">
    …
        <xs:element name="Processes">
            …
                <xs:element ref="Process" minOccurs="0" maxOccurs="unbounded"/>
…
</xs:element>

<xs:element name="Process">
    …
        <xs:element name="Activities">
            …
            <xs:element ref="Activity" minOccurs="0" maxOccurs="unbounded"/>
                …
        <xs:element name="Synchronizations">
            … …
        </xs:element>
    </xs:all>
    <xs:attribute name="uuid" type="xs:ID" use="required"/>
    <xs:attribute name="name" type="xs:anySimpleType"/>
</xs:complexType>
</xs:element>
…
<xs:element name="Activity">
    …
        <xs:element name="StartActivity">
            …
                <xs:element name="Precondition" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="uuid" type="xs:ID" use="required"/>
            <xs:attribute name="name" type="xs:anySimpleType"/>
            <xs:attribute name="enactmentState" type="xs:anySimpleType"/>
            …
        <xs:element name="SubflowActivity" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
            <xs:attribute name="uuid" type="xs:ID" use="required"/>
            <xs:attribute name="name" type="xs:anySimpleType"/>
            <xs:attribute name="process" type="xs:IDREF"/>
            <xs:attribute name="enactmentState" type="xs:anySimpleType"/>
            …
        <xs:element name="AtomicActivities" minOccurs="0">
                                    …
                                <xs:element name="InputParameters">
                                    …
                                <xs:element name="OutputParameters">
                                    …
                                <xs:element name="Performers">
                                    …
                                <xs:element name="Performances">
                                    …
                            <xs:attribute name="uuid" type="xs:ID" use="required"/>
                            … …
                        …
        </xs:element>
        <xs:element name="EndActivity">
            …
                <xs:element name="Postcondition" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="uuid" type="xs:ID" use="required"/>
            … …
    </xs:sequence>
    <xs:attribute name="uuid" type="xs:ID" use="required"/>
    <xs:attribute name="name" type="xs:anySimpleType"/>
    … …
</xs:complexType>
  </xs:element>
```

Figure 16. PRAISE definition tool

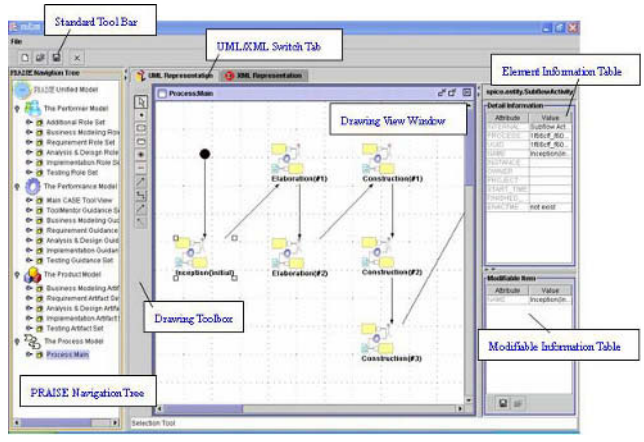


Figure 17. UML representation

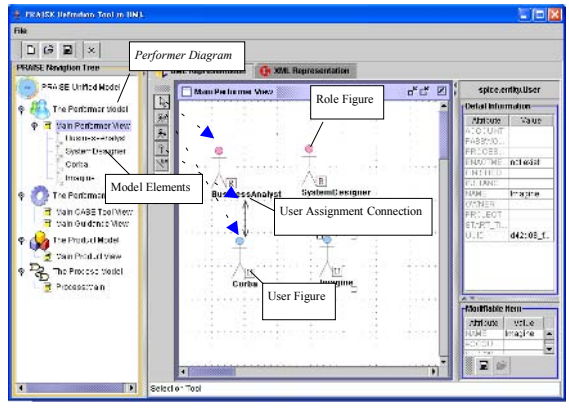


Figure 18. XML representation

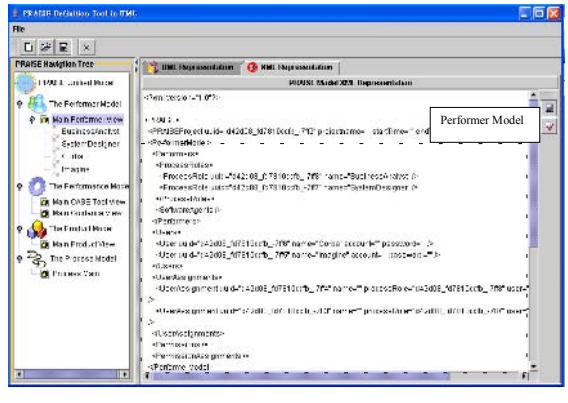


Figure 19. Create multiple views for the performer model

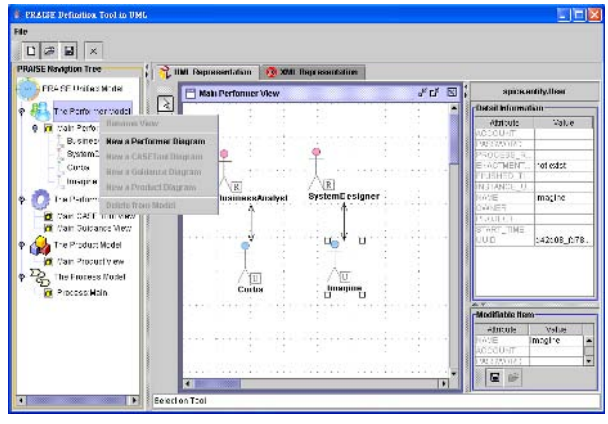


Figure 20. Modify the properties of mode elements

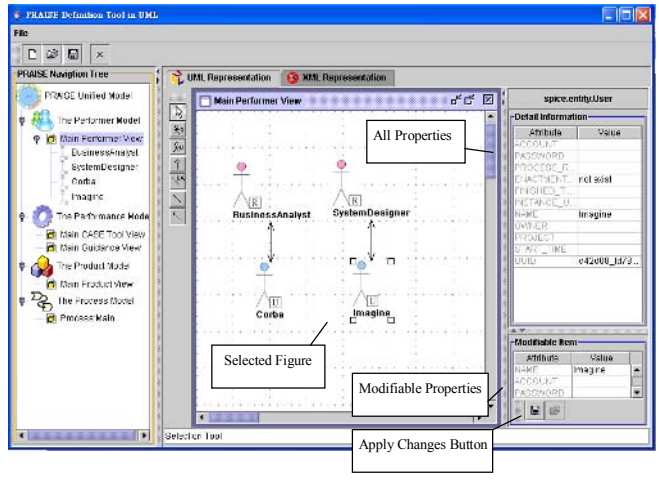


Figure 21. Rename views

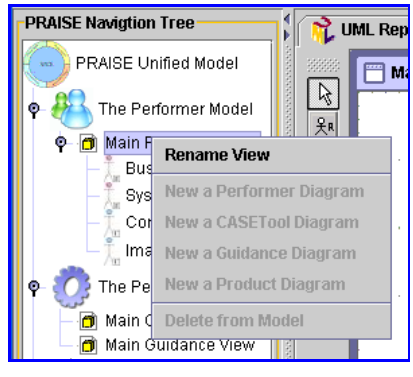


Figure 22. Delete model or view

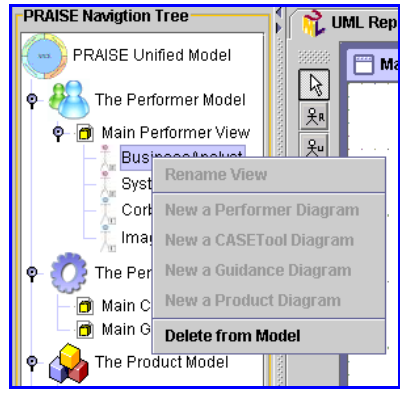


Figure 23. The CASE tools view

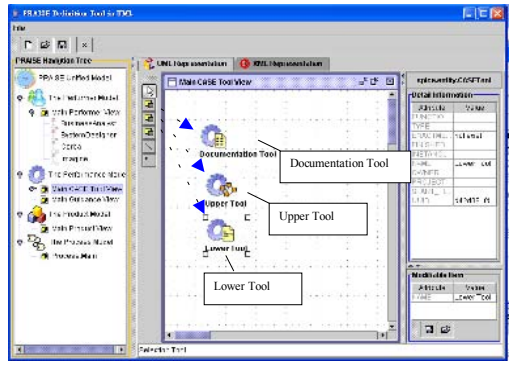


Figure 24. The guidance view

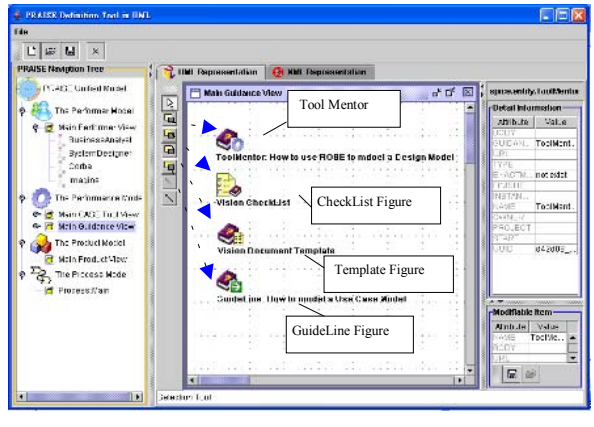


Figure 25. The development process of the library system

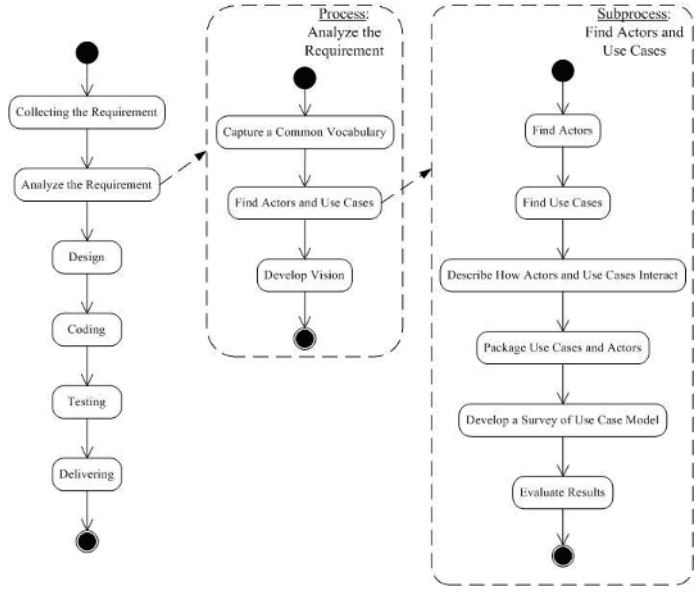
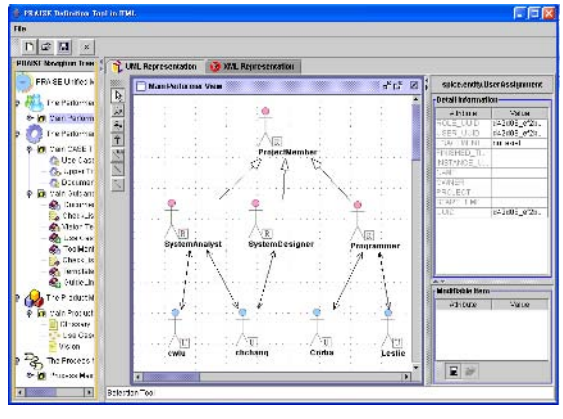


Figure 26. The performer view in PRAISE

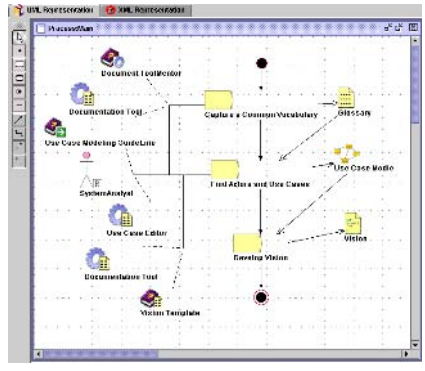


PRAISE Definition Tool

The PRAISE definition tool is used to assist software developer to define the performer model, performance model, product model, and process model. As shown in Figure 16, it provides a GUI-based window for a navigation tree for these four models, the UML tool, element information table, and modification information table.

The PRAISE definition tool contains the performer modeling tool, performance modeling tool, product modeling tool, and process modeling tool.

Figure 27. The corresponding representation of "Process: Analysis the Requirement" in PRAISE



Performer Modeling Tool

The performer modeling tool provides the facility to define each user's role and their relation with assigned tasks and other users using UML and is automatically represented as part of the PRAISE model in XML internally. Each defined performer model can be a template for later reuse, which is shown in *Performer View*. Figures 17 and 18 show the UML and XML representation, correspondingly. To support the software evolution activities, each modified performer modeling information will be tracked and shown in *Modifiable item window.* Since each performer may serve as different roles, the performer modeling tool can create, modify, rename, and delete views each performer model (see Figures 19, 20, 21, and 22).

Performance Modeling Tool

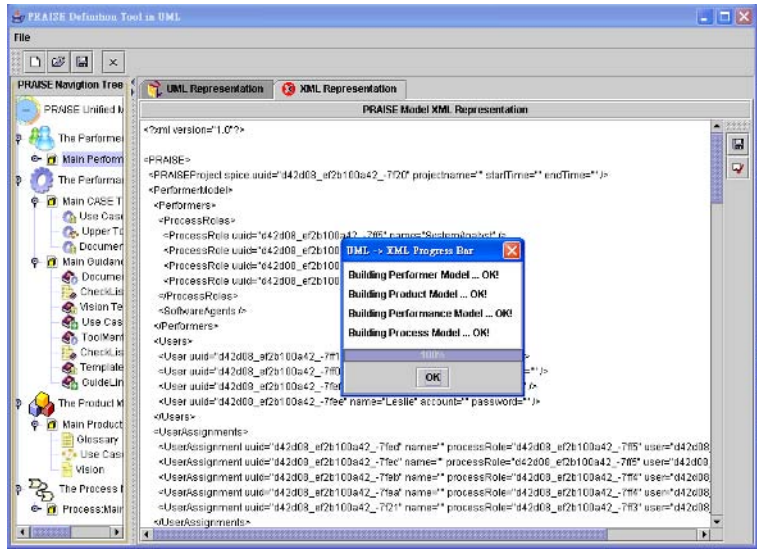
The Performance modeling tool offers an operating window for the integrated CASE tools, such as documentation tool, Rational Rose, and JBuilder (see Figure 23). It also provides the adopted guidance, such as tool mentor and checklist (see Figure 24).

An Example of Applying PRAISE to Software Evolution

To demonstrate the advantages of PRAISE, we have used PRAISE to assist the software development and evolution of a library system. The first step is to define the development process of this project. Its process is shown in Figure 25.

In PRAISE, the project manager defines the development process using the*PRAISE Definition Tool*. Figure 26 shows the definition of each performer (role) and the relations with other performers in this project. Figure 27 shows the modeling of the partial development process, "Requirement Analysis process". The process consists of defining the tasks of "Capture Common Vocabulary,", "Find Actor," and "Specify Use Case," and "Develop Vision." PRAISE provides the "Document ToolMentor," "Documentation

Figure 28. The corresponding XML representation of PRAISE model



Tool," "UML Tool," "UML Modeling GuideLine," and "Vision Template" tools to support these tasks. Software products such as "Glossary," "Use Case Model," and "Vision" will be generated by these tools after finishing these activities. The activities of process include roles, toolsets, process, and work products (Figure 27). The information of modeling and produced products are represented in the PRAISE model in XML format (see Figure 28).

To simplify the explanation of our example, we choose a subprocess, "Find Actors and Use Cases," to show the details of the model. Table 3 lists the related steps, participators, toolsets, and products of this subprocess.

The information listed in Table 3 can be modeled in the PRAISE model. Figures 29, 30, and 31 show the partial semantic of the process model, the performer model, and the performance model, respectively. In order to identity the elements, each element in the PRAISE model has a UUID whose attribute type is xs:ID as a unique identity. If an element needs to refer to others, we can use ID to refer to other elements.

For example, Figure 29 shows the modeling information of "Find Actor and Use Cases."

Some semantics are described as follows:

1. The <Performer_ref performer="PR002"> means that one Performer in this AtomicActivity refers to a ProcessRole element with the same UUID as PR002 in the Performer Model, that is, <ProcessRole uuid="PR002" name="SystemAnalyst"/>.

Table 3. The activities of subprocess: "Find Actors and Use Cases"

Process	Find Actors and Use Cases	
Pre-condition	Collect complete user requirements	
Post-condition	Produce the use case diagram	
Steps	Find Actors	
	Find Use Cases	
	Describe How Actors and Use Cases Interact	
	Package Use Cases and Actors	
	Develop a Survey of Use Case Model	
	Evaluate Results	
Participators	SystemAnalyst	
Tools	CASE Tool	PRAISE UML Modeling Tool
	Guidance	Use-Case Analysis Workshop
Products	Inputs	User Requirements
	Outputs	Use Case Models

2. The <Performance_ref uuid="FY0002"> and <Performance_ref uuid="GL002"> in the same AtomicActivity will respectively refer to <Functionality uuid="FY0002" name="Modeling Use Cases Model"/> or <CASETool uuid="CT002" name="PRAISE UML Modeling Tool"> <Guideline uuid="GL002" name="Use-Case Analysis Workshop"/> in the Performance Model.
3. The <Integration_link …> is used to link the components that share some semantic information in other models of the PRAISE model.

This means that the "Find Actor and Use Case" atomic activity will be performed by the "System Analyst" role and be supported with a guideline "Use-Case Analysis Workshop" and "Modeling Use Cases Model" functionality in "PRAISE UML Modeling Tool." The tool will use "Glossary" as the input product and finally will produce a "Use Case Model" with "Init" state. Similarly, each element can be referred to by other elements in the same way.

With the support of PRAISE, each participator will know what, when, and how to perform the assigned tasks. On the other hand, manager can control the progress of this project according to the PRAISE model, too. For example, in Figures 29, 30, and 31, System Analyst (uuid = "PR002") must do the Atomic Activity (uuid="AA0002") "Find Actors and Use Cases" according to input parameter uuid="IP0001," performance: "Modeling Use Cases Model" (uuid="FY0002"), and "Use Case Analysis Workshop" (uuid=

Figure 29. Partial semantic of the PRAISE process model

```
<ProcessModel>
 … …
      <AtomicActivity uuid="AA0002" name="Find Actors and Use Cases">
        <Integration_link xlink:lable = "AA0002" xlink:title="Find Actors and Use Cases"/>
        <Steps>
            <Step uuid="SP0003" name="Find Actors"/>
            <Integration_link xlink:from = "AA0002" xlink:to="SP0002"/>
            <Step uuid="SP0004" name="Find Use Cases"/>
            <Integration_link xlink:from = "AA0002" xlink:to="SP0004"/>
            <Step uuid="SP0005" name="Describe How Actors and Use Cases Interact"/>
            <Integration_link xlink:from = "AA0002" xlink:to="SP0005"/>
            <Step uuid="SP0006" name="Package Use Cases and Actors"/>
            <Integration_link xlink:from = "AA0002" xlink:to="SP0006"/>
            <Step uuid="SP0007" name="Develop a Survey of Use-Case Model"/>
            <Integration_link xlink:from = "AA0002" xlink:to="SP0007"/>
            <Step uuid="SP0008" name="Evaluate Your Results"/>
            <Integration_link xlink:from = "AA0002" xlink:to="SP0008"/>
        </Steps>
          … …
      <InputParameters>
            <Parameter inputProduct="IP0001" />
      </InputParameters>
        <OutputParameters>
            <Parameter outputProduct="OP0001" />
        </OutputParameters>
      … …
        <Performers>
            <Performer_ref performer="PR002"/>
            <Integration_link xlink:from = "AA0002" xlink:to="PR002"/>
        </Performers>
        <Performances>
            <Performance_ref performance="FY0002"/>
            <Integration_link xlink:from = "AA0002" xlink:to="FY0002"/>
            <Performance_ref performance="GL002"/>
            <Integration_link xlink:from = "AA0002" xlink:to="GL002"/>
            <Performance_ref performance="TM002"/>
            <Integration_link xlink:from = "AA0002" xlink:to="TM002"/>
        </Performances>
      </AtomicActivity>
… …
</ProcessModel>
```

Figure 30. Partial semantic of the PRAISE performer model

```
      <PerformerModel>
        <Performers>
            <Performer>
                <ProcessRoles>
                    … …
                    <ProcessRole uuid="PR002" name="SystemAnalyst"/>
                    <Integration_link xlink:lable = "PR002" xlink:title="SystemAnalyst"/>
                    <Integration_link xlink:from = "PR002" xlink:to="AA0002"/>
                    … …
      </PerformerModel>
```

"GL002"), and so forth. When this activity is accomplished, SystemAnalyst must generate product uuid="OP0001" as output.

After the project manager finishes the development process definition, the developer can log in PRAISE according to the PRAISE client application, Figure 32 is the user login picture of PRAISE Client Application. And then, PRAISE will show the task following PRAISE model in task table. Figure 33 shows a developer, "chchang," log in. There are two tasks have been assigned to him, which are shown in the task table. After he finishes this task, PRAISE will pass the next task to the corresponding assigned developer.

PRAISE also provides UML editor to modeling object-oriented analysis and design, where each icon and specified model information are supported by XUM. Figure 34 shows the work product of the process: "Find Actors and Use Cases." Figure 35 shows the corresponding XUM representation (Chu et al., 2002; Lu et al., 2003). Figure 36 shows the modeling of the class diagram.

Therefore, software development in PRAISE will get a strong support since it offers rich information and links related information together. First, we will show the tractability of the PRAISE model.

Assume that we want to modify the use case diagram in Figure 34. The PRAISE model lets us know the product semantics and related tools. Figure 37 shows the related semantics in different submodels that have been integrated and modeled in the PRAISE model. After the modification and analysis, the affected elements will be shown in the ripple effect manager (see Figure 38).

Usually, changes to software systems may involve software products produced by different tools in a different phase. Figure 39 shows an example that covers the execution

Figure 31. Partial semantic of the PRAISE performance model

```
<PerformanceModel>
  <CASETools>
      … …
      <CASETool uuid="CT002" name="PRAISE UML Modeling Tool">
          <Functionality uuid="FY0002" name="Modeling Use Cases Model"/>
              <Integration_link xlink:lable = "FY0002" xlink:title="Modeling Use Cases
Model"/>
              <Integration_link xlink:from = "FY0002" xlink:to="AA0002"/>
          <Functionality uuid="FY0003" name="Modeling Design Model"/>
              <Integration_link xlink:lable = "FY0003" xlink:title="Modeling Desing
Model"/>
            ……
      </CASETool>
  </CASETools>
  <Guidances>
      … …
      <Guideline uuid="GL002" name="Use-Case Analysis Workshop"/>
              <Integration_link xlink:lable = "GL002" xlink:title="Use-Case Analysis
Workshop"/>
              <Integration_link xlink:from = "GL002" xlink:to="AA0002"/>
      … …
</PerformanceModel>
```

Figure 32. PRAISE user login window

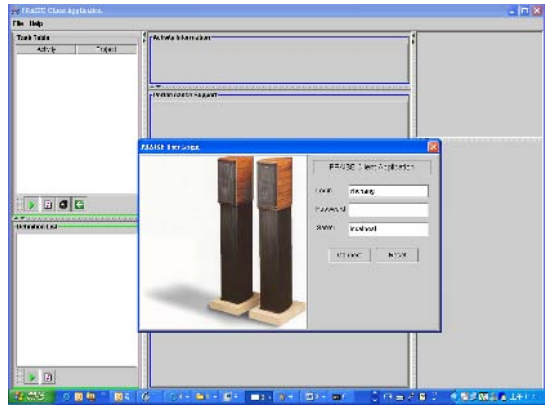


Figure 33. The task table

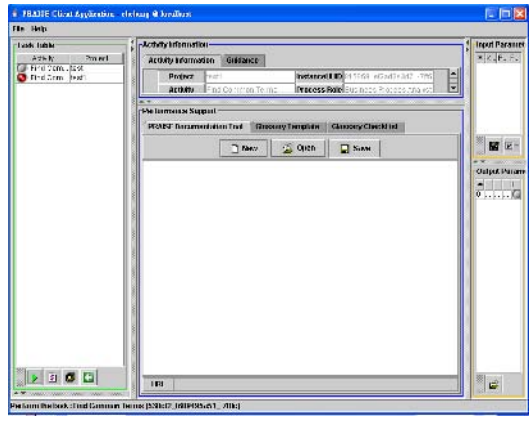


Figure 34. A partial use case diagram of a library system

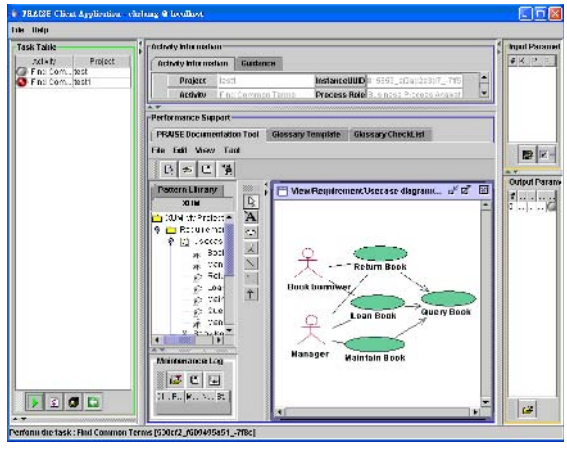


Figure 35. XUM Specification of the use case diagram

```
<ProductModel>
… …
      <Requirement>
        <UseCase_Daigram>
          <Integration_link xlink:lable = "OP0001" xlink:title="Use Case Diagram of library
      system"/>
          <Actor name="Book Borrower"/>
          <Actor name="Manager"/>
          <Usecase name="Loan Book">
            < Integration_link xlink:label="RA001" xlink:title="Use Case of Loan_Book" />
            … …
          </Usecase>
          <Usecase name="Return Book"> … …
          <Usecase name="Maintain Book"> … …
          <Usecase name="Query Book"> … …
          <Relationship from="Book Borrower" to="Loan Book" type="association"/>
          <Relationship from="Book Borrower" to="Return Book" type="association"/>
          <Relationship from="Manager" to="Loan Book" type="association"/> … …
        </UseCase_Daigram>
      </Requirement>
… …
</ProductModel>
```

Figure 37. The relationship between "Use Case: Return Book" and corresponding class diagram

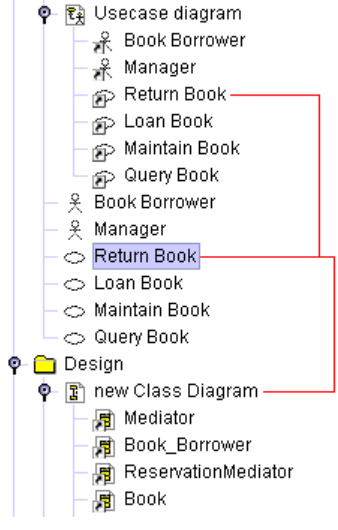


Figure 36. Corresponding class diagram of "Use Case: Return Book"

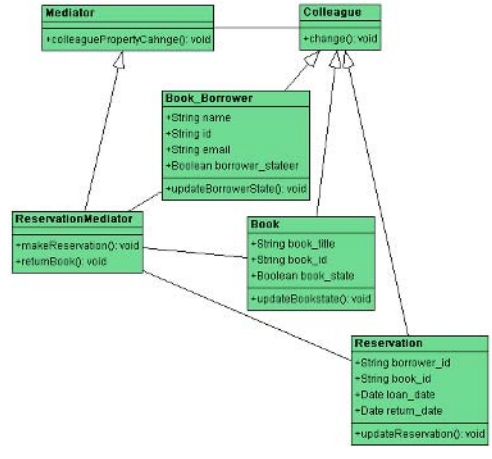


Figure 38. The ripple effect manager of PRAISE

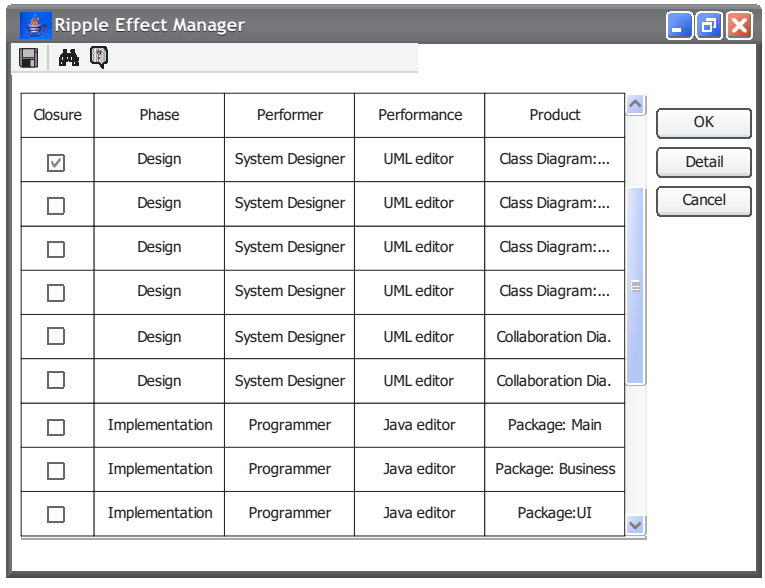


Figure 39. The execution steps of the ripple effect manager

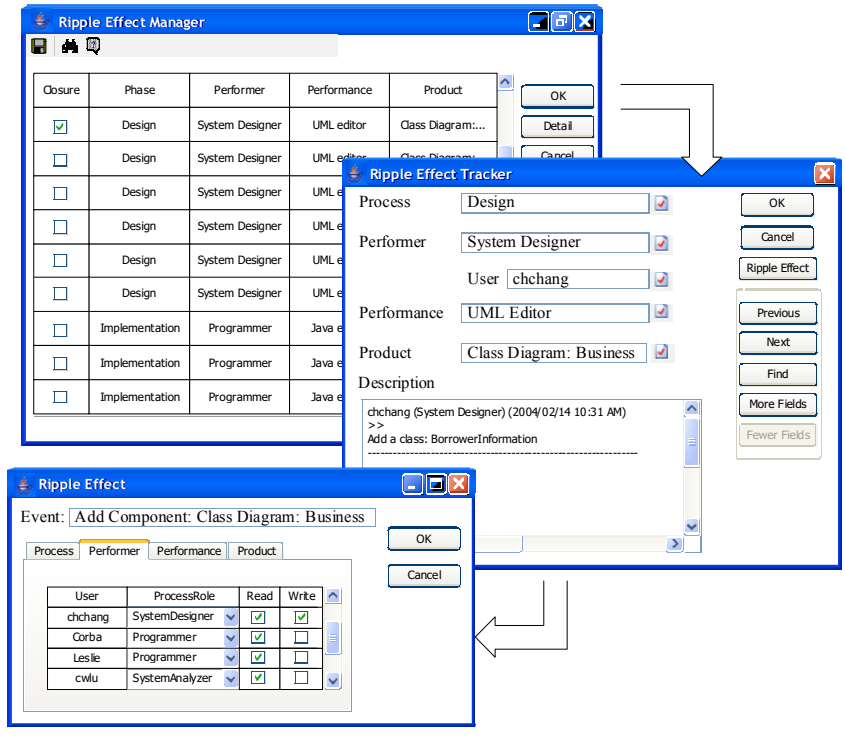


Table 4. Some statistics of the library system

Process	Performer	Performance	Product
Analysis /3 activities	System Analyzer / 1	UML/Use Case editor	2 use case diagram / 2 roles 11 use cases
Design/ 12 activities	System Designer/ 2	UML/ Class diagram, Collaboration diagram, Sequence Diagram, Component diagram, Statechart diagram editor	Logical View: Business / 5 classes Database / 1 class UI / 20 classes Utility / 1 classes 8 Class diagrams 10 Sequence diagrams 7 Collaboration diagrams 4 Statechart diagram
			Component View: Business / 5 Database / 1 UI / 19 Utility / 1 Main / 1 5 Component diagrams
Implementation/ 27 activities	Programmer/ 2	Java Editor/ JDK	27 files / 4100 (loc)
Testing / 11 activities	Tester/ 2		11 reports
Total 4 process/ 53 activities	Total 4 role/ 7 men	Total 8	Total modules: 141

of the ripple effect manager. With the support of PRAISE, the activities are organized and supported much more effectively.

To show the efficiency of the support from PRAISE, we have collected some statistics. We have used PRAISE to support the development of the library system which is a window-based system implemented by using Java language with 31 classes and a size of 4,100 lines of code. All information of process, performer, performance, and product are listed in Table 4.

Ripple Effect

When there is a need to modify a component in the product model, we must realize how many components that may be affected by this modification. Table 5 shows the scale of efforts required for this impact analysis with and without the PRAISE support.

We can see that the impact degree of PRAISE is smaller than that of the traditional approach. For larger projects, the difference between PRAISE and the traditional way becomes even larger.

Table 5. Impact degree of search related components

	Process	Performer	Performance	Product
PRAISE	(4 + 1) / (4 + 12)	(4 + 1) / (4 + 7)	5 / 8	(27 + 7) / (141 * 2)
	31.25 %	**45.45 %**	**62.5 %**	**12 %**
Tradition way	(4 + 12) / (4 + 12)	(4 + 2) / (4 + 7)	8 / 8	141 * 2
	100 %	**54.54 %**	**100 %**	**100 %**

Conclusion

In this chapter, we have introduced PRAISE, an XML-based metamodel for a process and agent-based integrated software development environment. PRAISE integrates and unifies a set of models with well-accepted software paradigms of a software development process into a unified model represented in XML.

PRAISE facilitates software evolution by exploiting the unification of models of the software development process, including roles, processes, toolsets, and work products. Its unification links, which connect components of models in the software development process, enable systematic software reuse for analysis and design at the earliest stages in the software life cycle.

PRAISE integrates software documents scattered over the process of development, while integrating phases of the software process itself. PRAISE helps users in the task of development all across the software process. PRAISE simplifies maintenance in all its aspects by upholding the connections of roles, processes, toolsets, and work products, while at the same time preserving the reset conditions of those in the views of the original paradigms and models. Users can update submodels of a system from any modeling techniques or paradigms as needed. Any implicit inconsistencies will present themselves in the other related models, allowing maintainers to deal with them in a clear, systematic manner.

The PRAISE approach improves software development and integrates the process, roles, toolset, and product into a unified model. PRAISE facilitates development by exploiting the unification and the common points of models. Its unification links, which connect components among four models, enable cross-phase tractability during software development. Users can update related elements of a software development process from any modeling or process as needed. According to this discussion and validation, we believe that the PRAISE approach proposed in this chapter can be extended and benefit more activities in various processes for software development.

Before we close, we would like to point to some future work on PRAISE. First involves extending PRAISE to support collaborative software development. The scale of software projects is so enormous that cooperation is necessary. Access control is the next target of PRAISE.

A second future study is needed to enhance PRAISE. Presently, it is just a prototype. A software environment that supports the PRAISE approach and provides more automation, guidance, distributed processing with collaborative work, and assistance to the activities of software evolution and maintenance is another important future task.

A third area for further study involves the PRAISE model-based analysis. Currently, most software products retain no structural representation. As a result, high-level information, such as roles, processes, and toolsets, has no explicit links to the corresponding. Representing the software development process information in a PRAISE model that is linked to the related elements/documents can facilitate the analysis. Because XML documents are represented in DOM and the compilers of XML are already available, efforts to implement analysis toolsets are much easier.

A fourth topic to study is an integrated software estimation model, like CMMI (Software Engineering Institute, 2004). Most of the estimative factors like product, process, and roles have been integrated into PRAISE. If CMMI can be integrated into PRAISE, the support of software development and evolution will be more complete.

References

Chu, C.W., Chang, C.H., Lu, C.W., Jiau, H.C., Yang H., Bing, Q., & Chung Y.C. (2002). Enhancing software maintainability by unifying and integrating standards. In *Advances in software maintenance management: technologies and solutions* (pp. 114-150). Hershey, PA: Idea Group Publishing.

Engels, G., Lewerentz, C., Nagl, M., Schäfer W., & Schürr, A. (1992). Building integrated software development environments. Part I: Tool specification. *ACM Transactions on Software Engineering and Methodology, 1*(2), 135-167.

Finkelstein, A., Kramer, J., & Nusibeh, B. (1994). *Software process modeling and technology*. New York: John Wiley & Sons.

Foundation for Intelligent Physical Agents. FIPA (2000, November). *Agent message transport envelope representation in bit efficient specification*, Specification number SC00088. Available online at *http://www.fipa.org/specs/fipa00088/.*

Habermann, A.N. & Notkin, D. (1986). Gandalf: software development environments. *IEEE Transactions on Software Engineering, 12*(12), 1117-1127.

Harrison, W., Ossher, H., & Tarr, P. (2000). Software engineering tools and environments: A roadmap. *Proceedings of International Conference on Software Engineering Proceedings of the conference on the future of Software engineering* (pp. 261-277). New York: ACM Press

Lu, C.W., Chu C.W., Chang, C.H., Lian, W.D., & Yang, D.L. (2003). Integrating drivers paradigms in evolution and maintenance by an XML-based unified model. *Journal of Software Maintenance and Evolution, 15*(3), 111-144.

Maes P. (1997). *General tutorial on software age*nts. Available online at *http://pattie.www.media.mit.edu*

Mi, P. & Scacchi, W. (1996). A meta-model for formulating knowledge-based models of software development. *Decision Support Systems, 17*(4), 313-330.

Object Management Group. (2002). *Meta Object Facility (MOF) Specification*, Version 1.4. Needham, MA: OMG.

Object Management Group. (2003). *XML Metadata Interchange (XMI) Specification*, Version 2.0. Needham, MA: OMG.

Object Management Group. (2003, April 2). *The software process engineering metamodel (SPEM),* Revised Submission, OMG document number: ad/2001-03-08. Retrieved May 16, 2003, from *http://www.omg.org/cgi-bin/ doc?ptc/2002-01-23*.

Ossher, H. & Harrison, W. (1990). Support for change in RPDE3. *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environment*. ACM Press. Reengineering methodology. *IEEE Transactions on Software Engineering*, *12*(2), 326-345.

Thompson, H.S. (2003). *W3C XML Pointer, XML Base and XML Linking.* The World Wide Web Consortium. Retrieved August 21, 2003, from *http://www.w3.org/XML/Linking*

Wassermen, A.I., Pricher, P.A., Shewmake, D.T., & Kersten, M.L. (1986). Developing interactive information systems with the user software engineering methodology. *IEEE Transactions on Software Engineering, 12*(2), 326-345.

Workflow Management Coalition (1994, November). *The Workflow Reference Model.* Document Number WFMC-TC-1003, Version 1.1.

Chapter VII

Developing Requirements Using Use Case Modeling and the Volere Template: Establishing a Baseline for Evolution

Paul Crowther, Sheffield Hallam University, UK

Abstract

A major contributor to the development of a quality final product is a complete, consistent, and detailed requirement specification (Pressman, 2000). No matter how good the specification and its translation into an initial system, it will evolve once it is released to users as the requirements and the environment change and the users develop. The aim of this chapter is to provide a method of establishing the baseline in terms of the requirements of a system from which evolution metrics can be effectively applied. UML (Rumbaugh, Jacobson, & Booch, 1999) provides a series of models that can be used to develop a specification which will provide the basis of the baseline system. This can then be used as a datum from which measurements can be made. One of the starting points for modeling is use case analysis. Other models can then be developed based on these initial models. One of the difficulties with this approach is

that once the initial models have been agreed upon, they are not maintained as the later more detailed models evolve. The methods described in this chapter show how this can be achieved and measured.

Introduction

This chapter discusses the establishment of a baseline from which to measure the evolution of a software system. The work described is based on the development of a system designed to deliver a collaborative learning environment on personal Web-enabled mobile computing devices called MOBIlearn (2000). Further, the components which make up the system are being developed by a series of teams which are distributed throughout Europe.

A system in constant use evolves. This is because of changes in:

- Requirements
- Environment
- User development

However, the evolution needs to be both controlled and measured. The foundation of the initial instance of the system is the requirements on which it is based. Feedback from users and refinements as the environment is more completely understood will lead individual components to evolve and hence the overall system itself to evolve.

UML has the advantage that it can be used in conjunction with a variety of development methodologies while providing a readily understandable set of diagrams. These are based on a series of interconnected models that range from use cases used to develop requirements through collaboration diagrams used to determine how the use cases will be implemented on to logical models which will form the basis of the final software.

In this chapter, the primary emphasis will be on use cases and their role in establishing the base requirements of the system. These will be discussed in terms of their relationship to the Volere template which adds control and referencing. Finally this will be tied into the UML component model and the use of XML in the resulting service-oriented architecture.

Background

The Need to Establish a Baseline for Software Evolution

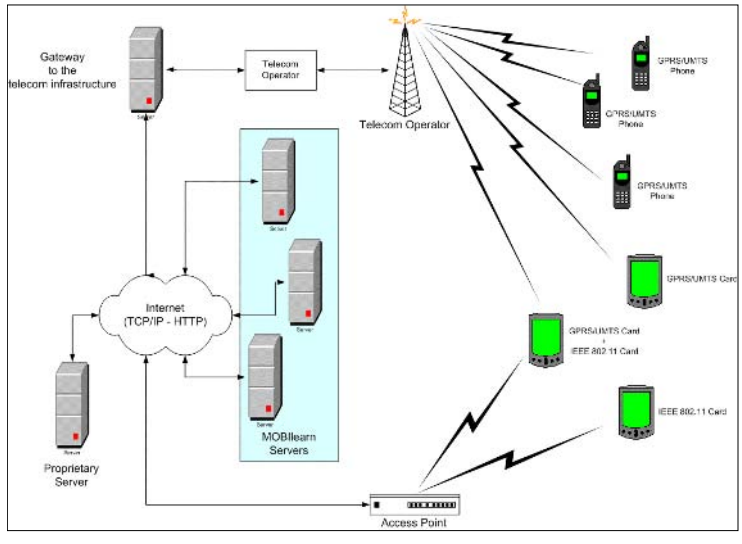
Software evolution is not a new study with references dating back to the 1960s (Lehman & Ramil, 2003). Evolution comes about because of the changes required to meet user or operational needs. Lehman (2000) has been a major contributor to studies of evolution with the FEAST projects where feedback (the F in FEAST) is the driver for the next evolutionary step.

The case study used in this chapter will be discussed in relation to using UML to prepare system specifications which establish a baseline from which the evolution of the software can be measured. Hall and Munson (2000) discuss metrics which can be applied to software and provide measures which can be applied to this baseline. What is needed is a method of creating the baseline in the first place. This is supported by Lehman's (2000) position paper where he defines S-type and E-type programs. Initially, programs should be of the specification (S) type giving a baseline. Once the programs are released and integrated into larger systems they become evolutionary (E-type).

The Case Study: MOBIlearn

This work is based on experiences from the MOBIlearn project funded by the European Framework V IST programme. Learners today want to learn when and where they want, in formal, nonformal, and informal ways (Brand, Petrak, & Zitterbart, 2002). MOBIlearn

Figure 1. MOBIlearn hardware architecture



meets learners requirements utilising mobile communications and personal computing devices such as PDAs, smart phones, and portable computers (see Figure 1).

A key part of the MOBIlearn project is the integration of new technologies in education. It aims at improving access to knowledge for selected target users giving them ubiquitous access to appropriate learning objects (Taylor, 2003). Initially, the Mobilearn requirements were provided by four scenarios:

- Master in Business Administration
- University orientation
- A European city famous for its art (Florence)
- Access to basic medical knowledge

The aim of MOBIlearn is therefore "…the creation of a virtual network for the diffusion of knowledge and learning via a mobile environment…to…demonstrate the convergence and merging of learning supported by new technology, knowledge management, and new forms of mobile communication." (MOBIlearn 2002, Annex 1, p. 7)

In the MOBIlearn project a series of distributed development teams have been established with specific roles or workpackages (WPs). One of these WPs is the development of requirement specifications to be used by the technical WP teams.

Initially, these requirements were derived from the user scenarios using use cases alone, one set for each scenario. The next stage was to amalgamate these into a single specification which could then be handed on to the software developers for the final system. The technical teams then developed a series of services which were required to service the use cases.

The development methodology for MOBIlearn was a combination of the service-oriented approach (SOA) and prototyping loosely based on Boehm, Egyed, Kwan, Port, Shah, and Madachy's (1998) spiral model. Modeling was done using the use case modeling parts of UML. The baseline is established as the first iteration of the spiral, referred to in MOBIlearn as the second prototype (the first prototype, as will be described in detail, was a throwaway demonstrator).

Volere Templates

Volere (Robertson & Robertson, 2001) is a template for requirement specifications. It provides a mechanism for documenting both functional and nonfunctional requirements as well as project constraints, drivers, and issues. Altogether there are 27 shells which can be completed to document a system's requirement. It should be noted that not all of the shells are mandatory, and modifications can be made to the shells. Several of the shell types were used in the MOBIlearn project including:

- Client and Stakeholder definition (shell type 2)
- Naming conventions and definitions (shell type 5)
- Relevant facts and assumptions (shell type 6)
- Functional and data requirements (shell type 9)

Basically the template was used as a form of data dictionary.

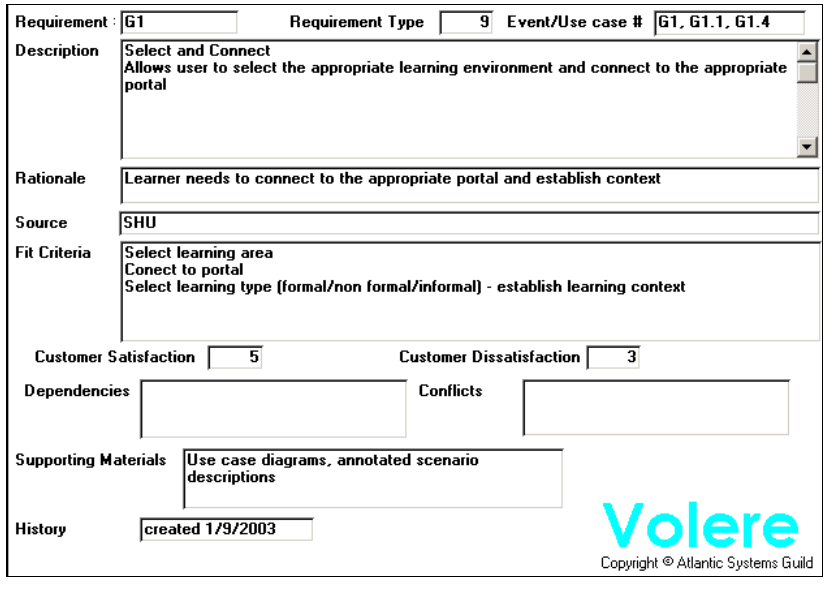
In the context of this chapter, it is the functional requirements and data requirement shell which are important. The functional and data requirement shells (shell type 9), an example of which is shown in Figure 2, are designed to specify the functional requirements that must be supported by the system. These shells are directly linked to use cases in the UML model.

Metrics and Baselines

Demeyer, Mens, and Wermelinger (2001) propose a software evolution benchmark for comparing various techniques dealing with software evolution. The MOBIlearn project meets the criteria proposed in terms of life cycle, evolution, and domain as a representative system — it is designed to be expanded beyond the existing four scenario domain.

The main thrust of this chapter is a baseline for metrics. There is much discussion of the best way of measuring software evolution. Mens and Demeyer (2001) make the distinction between predictive analysis and retrospective analysis. Predictive analysis attempts

Figure 2. Volere shell of type 9 (requirement)



to identify which parts of the system either need or are likely to be evolved and those that can suffer as a result of evolution. Secondly, there is retrospective analysis after or possibly during an evolving systems life.

Lehman (2000), on the other hand, considers evolution as a process that commences when an application moves from S-type to E-type. As a measure of evolution, it is the S-type that is the base. However, to be a suitable point from which to measure, there needs to be a series of features in place to help with the measurements. If these are in place, some of the analyses proposed by Mens and Demeyer (2001) may be unnecessary as evolution has been considered as part of the original design.

One problem with Lehman's (2000) approach, particularly when considered in terms of the MOBIlearn project, is that the development methodology is prototyping, so each prototype could be regarded as an evolution of the overall system, but the system could still be in S-type form as it had not moved from the original specification. On release, the system would almost certainly evolve as new domain applications are added (e.g., a theatre scenario).

Another approach is that described by Hall and Munson (2000) who see evolution as a series of discrete increments or builds. In their case, they are almost certainly considering systems rather than individual components. In their approach, they consider the application of metrics to measure evolution being applied more than once, stating "measuring an evolving software system only once can be very misleading." Hall and Munson (2000) also discuss the need for a measurement baseline as the basis for comparing the evolving system.

There is a certain dichotomy here between the different authors. Hall and Munson (2000) talk in terms of builds, that is, of a group of independent programs which are linked. The first build is the baseline, the second build, which may add or delete programs from the system, is the second build, or evolution, and so on. Lehman (2000), on the other hand, appears to be talking of individual programs. In this discussion, both points of view will be covered. Individual programs evolve from the initial specification and deployment as do integrated systems.

Establishing a Baseline

From the beginning, it was determined that the MOBIlearn system would evolve. The technical developers opted for a service-oriented approach (Grishikashvili, Badr, and Taleb-Bendiab, 2003), with the distributed teams taking on the development of a series of related services. The system requirements were developed independently and formed the basis of identifying the services.

The initial requirements were established in terms of a series of scenarios written as a storyline, for example, a group of individuals visiting a museum. These were developed by interviews with end users, both learners and administrators. These scenarios were then documented using use case diagrams. The scenarios were not exhaustive, and initially no attempt was made to determine generic use cases. This led to the system specification being:

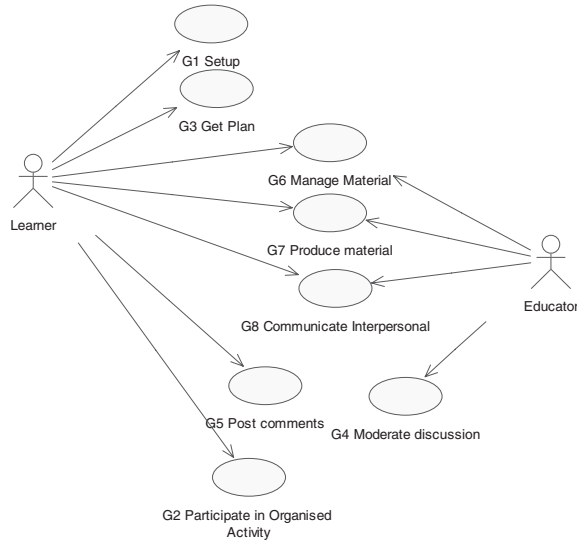
- too broad
- potentially incomplete
- inconsistent
- duplicated
- in effect, the specifications for three independent systems
- unsuitable for use as the basis of a system baseline.

These problems were solved by developing the requirements specification with a generic series of use cases. Templates were developed within the use cases' descriptions which were tied to shells within a Volere template.

UML, Volere, and Baselines

The Volere template provides a concise information base for the project and along with UML establishes a baseline from which the original S-type system (Lehman, 2000) is developed. As the system evolves, changed requirements can be tracked. The second prototype, which can be regarded as the measurement baseline, was produced based on these specifications. A third prototype and a release system are part of the project plan, and it is assumed that the system will continue to evolve after release. Although these are not strictly E-type in Lehman's terms, they are evolutionary cycles, and hence measurements can be applied.

Figure 3. Top-level generic or general use case diagram for MOBIlearn



Volere shells incorporate references to use case diagrams, but not the other way around. Likewise for every use case there must be a Volere shell, but a Volere shell may encompass several use cases. This has the advantage of combining use cases with similar functions. The main problem is overlap and redundancy between use case descriptions and fields in a Volere shell. The way around this was to adopt the Cockburn (1998) template for documenting use cases which was adapted for this project.

In MOBIlearn, the first stage of developing requirements was to develop use case diagrams for each of the four documented scenarios. Common features were then identified from these and a top-level generic use case diagram was constructed (see Figure 3).

Some of the top-level use cases were too general and need further elaboration. To this end, a second level of use cases was added; for example, Figure 4 shows the expansion of the G1 Setup use case.

To establish a common method of documenting all primitive (those that are not further expanded) use cases, the template suggested by Cockburn (1998) was used (see Figure 5). This overcame some of the shortcomings of fields in the Volere shell.

Establishing the requirements using UML and Volere is, however, only one part of establishing a baseline for system evolution. One could consider these to be the baseline requirements from which the S-type or first build or first iteration is produced. As part of the development process, UML was used to prepare a component diagram (see Figure 6) which met the requirements from use case diagrams and a Volere template. The use case and components were then linked together in a collaboration diagram. That is, each of the use cases was linked to the services that were required to implement it. This was found to be more effective in this project than a statechart approach such as that suggested by Ratcliffe and Budgen (2000). MOBIlearn is therefore a service-oriented architecture with a network addressable interface and strongly supporting interoperability.

Figure 4. Expansion of the generic G1 Setup use case

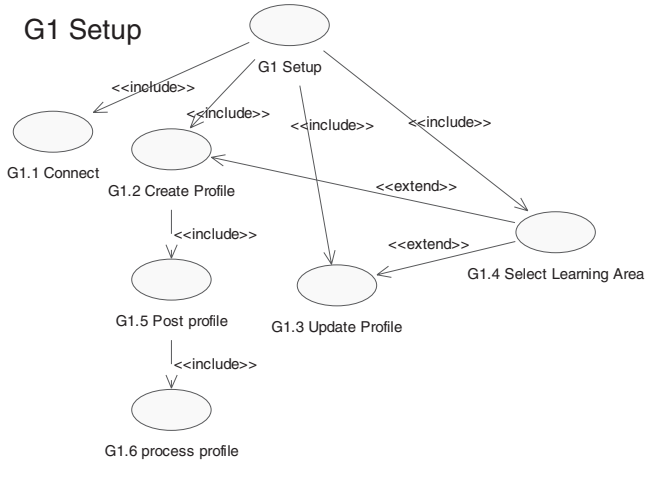


Figure 5. Use case description template based after Cockburn (1998)

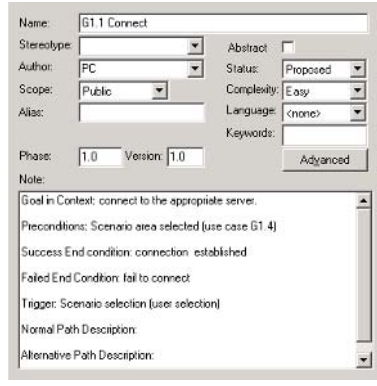


Figure 6. Fragment of the component diagram showing the relationships between services in MOBIlearn

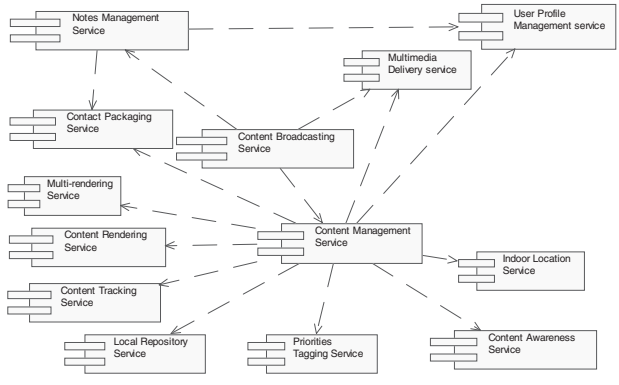


Table 1. Fragment of the of the service implementation plan

Service Name	Use Case	Service Code	Main work package (wp) involved	Other WPs involved	Prototype 2	Final System	Future
Portal		**PO**					
Portal Service	G1.1, G1.4	PO_POS	wp10	wp7, wp6	x	X	
User Registration Service	G1.1	PO_URS	wp10	wp7, wp6		X	
Login Service	G1.1	PO_LIS	wp10	wp7, wp6	x		
Authentication Service	G1.1	PO_ACS	wp10	wp7, wp6		X	
Authorization Service	G1.1	PO_AZS	wp10	wp7, wp6		X	
Billing Service	G1.1	PO_BIS	wp10	wp7			X
Service Discovery Service	G1	PO_SDS	wp10	wp7, wp6, wp8, wp9	x	X	
Content Delivery Service	G1	PO_CDS	wp10	wp7	X		

From the collaboration diagram a table was produced representing:

- The use cases
- Their related services
- The phase in which they would be implemented

This was regarded as a useful summary rather than having a relatively large number of individual collaboration diagrams. The service implementation was undertaken by a variety of different groups with integration planned as the second prototype. The first prototype was a throwaway demonstrator loosely based on the museum scenario before detailed use cases were developed. It could not be considered the S-type and the basis of the evolution of the system. The second prototype, however, is tightly coupled to the models and will be regarded as the S-type or baseline system from which the evolution of the system will be measured.

Table 1 shows the relationship between services and the generic use cases. An expansion of this table included a detailed analysis of services required for specific scenarios. For example, the heath scenario had use cases not found in the other scenarios and required some special services.

As part of the development and integration of the services, XML was used to exchange information between the various service components. The service-oriented approach was essential because of the distributed nature of the development teams and the final distributed nature and requirements of MOBIlearn.

Recommendations

Establishing a Baseline for Evolution

Pressman (2000) has provided useful and comprehensive templates for both software and system requirements specification. Generally, these involve a complete set of UML models and can be used to establish a baseline for measuring evolution. A combination of a Volere template and UML models cover most of the items in this template. One of the advantages of using a Volere template is that shells can be created to specify metrics to be used to measure the evolving system (see Figure 7). Note that the type 5 shell of the Volere template is used here. This is primarily for establishing definitions.

To develop a baseline, the following steps were used in the MOBIlearn project:

- Create a series of use scenarios.
- Develop use cases for the scenarios.

- Identify generic use cases for the integrated system.
- Create Volere shells for metrics definition.
- Create Volere shells for requirements from the use cases.
- Create component diagram (in terms of services).
- Correlate services to use cases using a collaboration diagram.

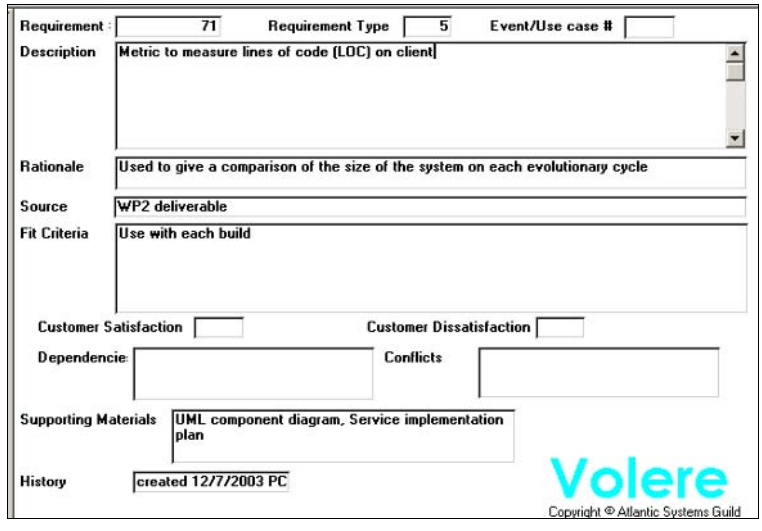
As new or changed requirements are found, in other words, the system requirements evolve, the affected Volere shells are updated. A change audit trail is built into the shell so the evolution of the requirements can be traced.

Problems Still to be Addressed

The main problem encountered during the MOBIlearn project was with control of the Volere template. Too many people were adding requirements, and there was too little control over standards and redundancy (duplicate entries). Management of the process was not trivial and has been improved. Future projects will have more stringent management and filtering processes in place for Volere.

There is also a requirement for a specialised Volere shell which is used for metrics. The shell currently used is for all definitions in the project (shell type 5); this is not an ideal situation. Therefore, it is proposed that a metric definition shell be added to the Volere template.

Figure 7. Volere shell defining a metric



Further Work

Once a baseline has been established, the next question is when to acquire the data for the metrics which have been defined. Establishing what is an evolutionary cycle requires more work. Do we consider each modification to each component of the overall system an evolutionary cycle? If this is so, the number of evolutions of a system could be quite large. Alternatively, a higher level view could be taken where each release of the system is a cycle. Finally, a combination of these two approaches may be feasible where evolution of individual software components is considered at one level, and evolution of the total in terms of major releases is considered, the Hall and Munson (2000) view, at the higher level. This has implications for the metrics which need to be defined and the design of the proposed metric definition Volere shell. As an interim measure, it is suggested that the level at which evolution is being measured is explicitly stated.

For example, in the case of MOBIlearn, new combinations of components (services in the MOBIlearn terminology) result in the release of a new evolution. Below this is the evolution of individual components. In this chapter, the emphasis has been on the evolution of the overall system.

Conclusion

The advantage of the methods described here is the establishment of a baseline and metrics at the beginning of the project. This has been developed at the system level, although the techniques could be applied at the component level. The basis of establishing the baseline is the development of generic use cases form a series of user centred scenarios. We are looking at the system level here. It was possible to develop the first iteration by selecting a subset of services to be implemented. This gave a representative version which could be used for enhancement in the second prototype. The service-oriented approach is such that third party products can be added. This, however, raises the question: Should a change in such a third party product be considered as an evolutionary cycle?

Acknowledgments

This work was part of MOBIlearn, a European framework F Information Society Technologies Program. I would like to acknowledge the contribution of all the MOBIlearn consortium members, particularly, those involved in WP 2 and WP 10.

References

Boehm, B., Egyed, A., Kwan, J., Port, D., Shah, A., & Madachy, R. (1998). Using the winwin spiral model: A case study. *IEEE Computer, 31*(7), 33-44.

Brand, O., Petrak, L., & Zitterbart, M. (2002). Support for mobile learners in distributed space, Learning Lab Lower Saxony (L3S). Retrieved July 12, 2003, from *www.learninglab.de/~brand/Publications/elearn02.pdf*

Cockburn, A. (1998). Basic use case template. Retrieved September 26, 2003, from *http://members.aol.com/acokcburn*

Demeyer, S., Mens, T., & Wermelinger, M. (2001). Towards a software evolution benchmark. *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE 2001),* Vienna, Austria.

Grishikashvili, E., Badr, N., & Taleb-Bendiab, A. (2003, June). Service-oriented approach for distributed application assembly and management. *Proceedings of PGNET 2003,* Liverpool.

Hall, G.A. & Munson, J.C. (2000). Software evolution: Code delta and code churn. *The Journal of Systems and Software, 54,* 111-118.

Lehman, M.M. (2000). Rules and tools for software evolution planning and management – position chapter. *Proceedings of FEAST Workshop,* Imperial College, London.

Lehman, M.M. & Ramil, J.F. (2003). Software evolution – background, theory, practice. *Information Processing Letters, 88,* 33-44.

Mens, T. & Demeyer, S. (2001). Evolution metrics. *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE 2001),* Vienna, Austria.

MOBIlearn (2002). *Next Generation paradigms and interfaces for technology supported learning in a mobile environment exploring the potential of ambient intelligence,* Annex 1, Information Society Technologies Program EU Proposal/Contract: IST-2001-37187.

Pressman, R.S. (2000). *Software engineering: A practitioner's approach, European Adaption* (5th ed.). London: McGraw-Hill.

Ratcliffe, M. & Budgen, D. (2001). The application of use case definitions in system design specification. *Information and Software Technology, 43,* 365-386.

Robertson, J. & Robertson, S. (2001). *Volere: Requirements specification template* (8th ed.). London: Atlantic Systems Guild.

Rumbaugh, J., Jacobson, I., & Booch, G. (1999). *The unified modelling language reference manual.* Reading, MA: Addison Wesley.

Taylor, J. (2003, May). A task-centred approach to evaluating a mobile learning environment for pedagogical soundness. *Proceedings of MLEARN 2003; Learning with Mobile Devices,* London.

Chapter VIII

Formalizing and Analyzing UML Use Case View Using Hierarchical Predicate Transition Nets

Xudong He, Florida International University, USA

Abstract

Unified Modeling Language (UML), developed by a group of leading experts in object-oriented methodologies, has become the standard object-oriented development methodology in the software industry. UML contains a set of diagrams for describing different views and aspects of systems. UML use case diagrams are used during requirements analysis to define a use case view that constitutes a system's functional model. Each use case describes a system's functionality from a user's perspective. However, the use case descriptions are often informal, which are error-prone and cannot be formally analyzed to detect problems in user requirements or errors introduced in a system functional model. A well-defined use case view is not only necessary for subsequent correct system design and implementation but also serves as a basis for future system evolution. Therefore, it is extremely important to ensure the correctness of the functional model captured in a use case view. In this chapter, we present an approach to formally translate a use case view into a formal model in hierarchical predicate transition nets that support formal analysis and thus are capable to detect possible requirements and modeling errors in a use case view.

Introduction

Software evolution is greatly impacted by a software design paradigm and software development methodologies. The object-oriented (OO) software development paradigm has been widely adopted in software industry in recent years and thus dictates the future software development and evolution. Among the various existing OO methods, UML (Booch, Rumbaugh, & Jacobson, 1999; Rumbaugh, Jacobson, & Booch, 1999) has become the de facto standard OO design language. UML contains a set of graphical notations for different views and aspects of software systems. While UML graphical notations are conceptually sound, they lack precise semantics and thus do not support formal analysis. Although it is desirable to separate and individually define the different aspects of a system, it is often not clear how to do and is very difficult to relate the different views together and to ensure their consistency.

To tackle the impreciseness of UML, there have been considerable research activities to make the UML semantics more precise in recent years. There is a worldwide pUML (precise UML) research group. Researchers have attempted to define formal semantics for class diagrams (Shroff & France, 1997; Evans, 1998; France, Evans, Lano, & Rumpe, 1998; Lano & Bicarregui, 1998; Kim & Carrington, 1998; McUmber & Cheng, 1999; He, 2000a; McUmber & Cheng. 2001), use case diagrams (Overgaard & Palmkvist, 1998, Back, Petre, & Paltor, 1999; He 2000b), interaction diagrams Knapp 1999, and statechart diagrams (McUmber & Cheng, 1999; Saldhana & Shatz, 2000; Saldhana, Shatz & Hu et al., 2001; Dong & He, 2001; McUmber & Cheng, 2001; Dong, Fu, & He, 2003). A variety of formal methods have been applied in the previous attempts, including variants of Z, variants of logic and temporal logic, refinement calculus, and variants of Petri nets.

In this chapter, we focus on one of the UML notations, *use case diagrams*. Use case diagrams were proposed by Jacobson, Christerson, Jonsson, and Overgaard (1992) to capture typical system use scenarios in system analysis. UML use case diagrams are used to define a use case view that constitutes a system's functional model. Each use case documents a system's functionality from a user's perspective. Use case analysis is one of the major activities of system requirements analysis and forms the backbone of the unified software development process (Jacobson, Booch, & Rumbaugh, 1999). A well-defined use case view is not only necessary for subsequent correct system design and implementation but also serves as a basis for future system evolution. Currently, only informal semantics of UML use case diagrams exists (UML, 2003). A use case diagram depicts the relationships between use cases and actors as well as relationships between use cases. There is no standard language for defining use cases, although a variety of choices, including plain text, state machines, operations, and interaction diagrams, is suggested; hence, informal plain text is often used in defining use cases. The lack of a precise standard language for defining use cases and their relationships makes the understanding and realization of use case diagrams difficult and the formal analysis of use case diagrams impossible. As a result, many errors due to incomplete and inconsistent requirements as well as incorrect specification cannot be detected and revealed until late in the system development process.

Hierarchical predicate transition nets (HPrTNs) are chosen to provide a formal semantics for UML use case diagrams for the following reasons: (1) HPrTNs (He, 1996) are a

hierarchical high-level extension to Petri nets, which have a huge worldwide researcher and user community; (2) HPrTNs have a graphical representation that can help preserve the visual structure depicted by UML use case diagrams; (3) HPrTNs have an operational semantics that can precisely define the behaviors implied in a use case view; and (4) we have extensive experience in developing and using HPrTNs (He & Lee 1991; He & Yang, 1992; He, 1995; He, 1996; He, 1998; He, 2000c; He & Ding, 2001), and have obtained some preliminary results in formalizing UML notations using HPrTNs (He, 2000a & b; Dong & He, 2001; Dong et al., 2003).

In this chapter, we first introduce the essential concepts and notations of the use case diagrams and HPrTNs; then, we provide an approach to translate a use case view into an HPrTN; and finally, we illustrate how to analyze system properties using HPrTNs.

Use Case Diagrams

The use case view captures the external behavior of a system or a subsystem, and divides the system's functionality into transactions meaningful to system users. In UML, a use case view is represented in use case diagrams as a result of requirement analysis. The use case view is used as a basis for designing a subsequent system behavioral model represented in UML statechart diagrams and interaction diagrams.

The Basic Concepts and Notation

A use case "is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor" (Booch et al., 1999). A use case is represented by an *ellipse*. An actor, represented by a *stick figure*, denotes a role or an external environment that interacts with a system. A use case is connected to some actor(s) through solid line(s) called *association*.

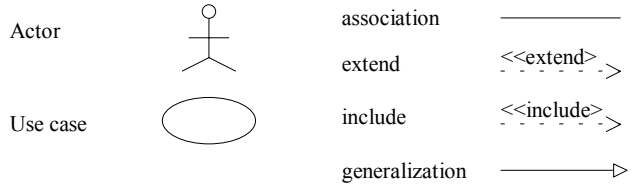
A use case (the base case) can include other use cases that represent some common and shared behavior. The include relationship is denoted by a dashed arrow from the base use case to the included use case with a stereotype label <<include>>.

A use case (the base case) can be behaviorally extended by another use case. The extension is implicit and extends the original behavior of the base case. The extend relationship is denoted by a dashed arrow from the extending use case to the base use case with a stereotype label <<extend>>.

A use case can have sub-use-cases. The generalization is represented by a solid line with a triangle head from a general use case to a specific use case.

Figure 1 summarizes the symbols of use case diagrams.

Figure 1. The symbols of use case diagrams



Informal Semantics

In document UML 2003, an informal semantics in English was provided for use case diagrams. The majority of the description cannot be considered as a semantic definition; rather, it is an informal explanation and discussion. We quote some of the description of relevant semantics below.

- **Actor:**

"Actors model parties outside an entity, such as a system, a subsystem, or a class, which interact with the entity. Each actor defines a coherent set of roles the entity can play when interacting with the entity. Every time a specific user interacts with the entity, it is playing one such role."

"Since an actor is outside the entity, its internal structure is not defined but only its external view as seen from the entity. Actor instances communicate with the entity by sending and receiving message instances to and from use case instances."

"Two or more actors may have commonalities, that is, communicate with the same set of use cases in the same way. The commonality is expressed with generalizations to another (possibly abstract) actor, which models the common role(s)."

- **Use Case:**

"A use case describes the interactions between the users and the entity as well as the responses performed by the entity." "A use case also includes possible variants of this sequence (e.g., alternative sequences, exceptional behavior, error handling, etc.)."

"A use case can be described in plain text, using operations and methods together with attributes, in activity diagrams, by a state machine, or by other behavior description techniques, such as preconditions and postconditions."

"A use-case instance is a performance of a use case initiated by a message instance from an instance of an actor. As a response, the use-case instance performs a sequence of actions as specified by the use case, like communicating with actor instances, not necessarily the initiating one."

"In the case where subsystems are used to model the system's containment hierarchy, the system can be specified with use cases at all levels." "A use case specifying one model element is then refined into a set of smaller use cases."

"Commonalities between use cases can be expressed in two different ways: with generalization relationships or include relationships. A generalization relationship between use cases implies that the child use case contains all the attributes, sequences of behavior, and extension points defined in the parent use case, and participate in all relationships of the parent use case."

"An include relationship between two use cases means that the behavior defined in the target use case is included at one location in the sequence of behavior performed by an instance of the base use case."

"An extend relationship defines that a use case may be augmented with some additional behavior defined in another use case." "The extend relationship contains a condition and references a sequence of extension points in the target use case." "The description of the location references by an extension point can be made in several different ways, like textual description of where in the behavior the addition should be made, pre- or post-conditions, or using the name of a state in a state machine."

There are several points worth of notice in the quotations of UML semantic descriptions:

(1) An actor is an external entity and thus no further detailed description is needed.
(2) A use case describes an interaction between the users and the entity (a system or subsystem).
(3) Many types of languages can be used to define use cases, ranging from informal plain text to formal graphical notations such as state machines.
(4) Several relationships can exist between use cases, which are depicted in use case diagrams.
(5) A use case instance is initiated by a message instance from an actor instance.

These points serve the basis for our translation approach.

An Example of Use Case Diagrams

Let us look at a simple library system that has two types of users — ordinary users and librarians. Ordinary users can use the system to search for books, and librarians use the system to process ordinary user requests for borrowing and returning books and to update library collections.

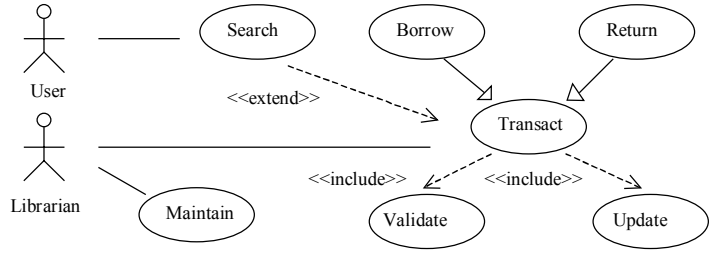
Based on the previous brief description, we can identify two actors — User for ordinary users and Librarian for librarians. We also can identify the following use cases: Search for User, Transact and Maintain for librarians. A further analysis of the library system reveals additional details of system behavior, and the refined use case diagram is shown in Figure 2.

The previous use case diagram specifies that the use case Search extends the use case Transact, which consists of two subtype use cases Borrow and Return, and includes two sub-use-cases Validate (user identification) and Update (book status).

Although a variety of techniques are suggested for describing use cases in UML 2003, the plain text description often is the choice due to its simplicity. The use cases in this example (except Maintain for simplicity) are defined using plain text as follows:

- **Search:** A user provides a book title and receives a message indicating whether the book is in the library or not.
- **Validate:** This case is invoked in the beginning either in borrow or return. It returns a value indicating either an invalid user or a valid user.
- **Update:** This use case is invoked either in borrow or return after user validation and book status is checked. It updates both the user record and the book record if all the conditions are satisfied.
- **Borrow:** This use case is initiated when a borrowing request is received. It invokes validate use case and searches for the requested title. It invokes update to change the user and book records.
- **Return:** This use case is initiated when a returning request is received. It invokes validate use case and checks the validity of the returned title. It invokes update to change the user and book records.

Figure 2. A use case diagram



The plain text description is easy to understand; however, it is often incomplete and ambiguous due to the inherent problems of a natural language. To detect and prevent the errors in requirements and specification, a more formal treatment and an analysis of these descriptions are necessary. This example is used to demonstrate the formalization and analysis approach in a later section.

Hierarchical Predicate Transition Nets

HPrTNs are a class of hierarchical high level Petri nets (He 1996), which extend predicate transition nets (Genrich, 1987) with super nodes (dashed circles and boxes for super places and transitions, respectively) and compound arc labels. The development of HPrTNs was inspired by the statecharts (Harel, 1988) and data flow diagrams (Yourdon, 1989). With the hierarchical constructs, a large net can be made modular and presented at different abstraction levels; and thus achieve the benefits of better understandability and modeling scalability. In this subsection, a simple description of HPrTNs is given. We try to make the description as understandable as possible by avoiding overwhelming number of mathematical symbols and rules. A formal method does not need to appear formal or complicated as long as it has a precise and well-defined syntax and semantics. A more detailed formal treatment can be found in He (1996).

The Syntax and Static Semantics of HPrTNs

An HPrTN N is a tuple $(P, T, F, \rho, SPEC, \varphi, L, R, M_0)$ consists of (1) a finite hierarchical net structure (P, T, F, ρ) – the *syntactic domain*, (2) an algebraic specification $SPEC$ — the *semantic domain*, and (3) a net inscription (φ, L, R, M_0) — the mappings from syntactic domain to the semantic domain.

(P, T, F) is the essential net structure, where $P \cup T$ is the set of nodes satisfying the condition $P \cap T = \varnothing$. P is called the set of *places* and T is called the set of *transitions*. There are two kinds of nodes for both places and transitions — *elementary nodes* (represented by solid circles or boxes) and *super nodes* (represented by dotted circles or boxes). In particular, we identify two subsets $IN \subseteq P \cup T$ and $OUT \subseteq P \cup T$ such that IN contains the heads of all incoming *non-terminating arcs* (an arc inside a super node is a non-terminating arc if one of its end is connected to the boundary of the super node) and OUT contains the tails of all outgoing non-terminating arcs. Nodes in $IN \cup OUT$ are called *interface* nodes. We use $\bullet IN$ to denote the set of the presets of all elements in IN, that is, $\bullet IN = \{\bullet n \mid n \in IN\}$; and $OUT\bullet$ to denote the set of the post-sets of all elements in OUT. F is the set of arcs and is called the *flow relation* satisfying the conditions: $P \cap F = \varnothing$, $F \cap T = \varnothing$, and $F \subseteq (\bullet IN \times IN \cup P \times T \cup T \times P \cup OUT \times OUT\bullet)$. An arc f can be uniquely identified by a pair of nodes (n_1, n_2) denoting its source and sink, in which n_1 (n_2) may denote the preset (post-set) of n_2 (n_1) when f is a non-terminating arc. An arc in an HPrTN

may represent a cluster of flows due to the use of super nodes, and the arc label is used to distinguish individual component flows.

ρ: $P \cup T \rightarrow \wp(P \cup T)$ is a hierarchical mapping that defines the hierarchical relationships among the nodes in P and T.

The underlying specification $SPEC = (S, OP, Eq)$ consists of a signature $\mathbf{S} = (S, OP)$ and a set Eq of S-equations. Signature $\mathbf{S} = (S, OP)$ includes a set of sorts S and a family $OP = (OP_{s_1,...,s_n, s})$ of sorted operations for $s_1, ..., s_n, s \in S$. For each $s \in S$, we use CON_S to denote $OP_{,S}$ (the 0-ary operation of sort s), that is, the set of constant symbols of sort s. The S-equations in Eq define the meanings and properties of operations in OP. We often simply use familiar operations and their properties without explicitly listing the relevant equations. $SPEC$ is a meta-language to define the sets of tokens $MCON_S$ (*ground terms* of the algebra), labels $Label_S(X)$, and constraints $Term_{OP,bool}(X)$ (terms of Boolean values) of an HPrTN.

The net inscription (φ, L, R, M_0) associates each graphical symbol of the net structure (P, T, F, φ) with an entity in the underlying $SPEC$, and thus defines the static semantics of an HPrTN.

- φ: $P \rightarrow \wp(S)$ is a sort assignment, which associate each place p in P with a subset of sorts in S. The sorts of elementary places are members of S in $SPEC$. The sort of a super place is defined as the union of sorts of its interface child places.
- L: $F \rightarrow Label_S(X)$ is a sort-respecting labeling of N. All simple labels of a compound label must have distinct identifiers, and all simple labels of arcs connected to the same node must have distinct identifiers.
- R: $T \rightarrow Term_{OP,bool}(X)$ is a well-defined constraining mapping of N, which associates each transition t in T with a first order logic formula defined in the underlying algebraic specification.
- M_0: $P \rightarrow MCON_S$ is a sort-respecting initial marking of N, which assigns a multi-set of tokens to each place p in P. The tokens of a super place are a sorted union of the tokens of its interface child places since only those tokens are externally accessible.

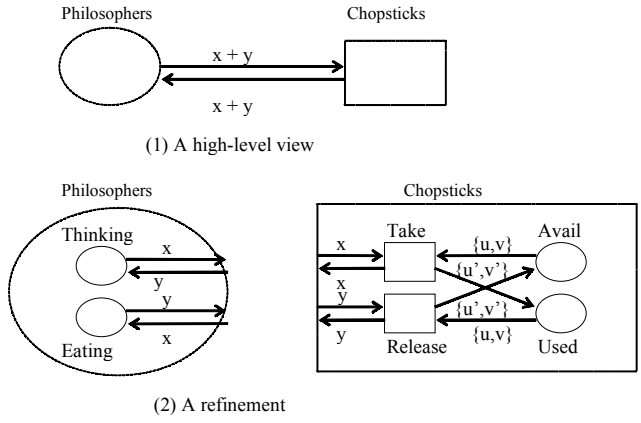
Dynamic Semantics

The dynamic semantics of an HPrTN is defined by its flattened predicate transition net (He, 1996). Thus, we can define the dynamic semantics of an HPrTN by only considering elementary places and transitions.

A marking M of an HPrTN is a mapping $P \rightarrow MCON_S$ from the set of elementary places to multi-sets of tokens. An elementary transition $t \in T$ is *enabled* in marking M if its preset $\bullet t$ contains enough tokens and its constraint $R(t)$ is satisfied with an *occurrence mode* α (a substitution of variables with tokens in the constraint). The statement can be formulated as follows:

$$\forall p \in \bullet t.(M(p) \supseteq L(p,t):\alpha) \wedge R(t):\alpha$$

Figure 3. An HPrTN specification of dining philosopher problem



The *firing* of an enabled elementary transition t in marking M with occurrence mode α produces a new marking M' defined by:

$$\forall p \in P.(M'(p) = M(p) - \{(L(p,t):\alpha\} \cup \{(L(t,p):\alpha\})$$

where $L(p, t) = \varnothing$ if $(p, t) \notin F$ and $L(t, p) = \varnothing$ if $(t, p) \notin F$.
Two transitions (including the same transition with two different occurrence modes) can fire concurrently if they are not in *conflict* (the firing of one of them disables the other). Conflicts are resolved nondeterministically. The firing of an elementary transition is atomic. We define the behavior of an HPrTN to be the set of all possible maximal execution sequences containing only elementary transitions. Each execution sequence represents consecutively reachable markings from the initial marking, in which a successor marking is obtained through a step (firing of some enabled transitions) from the predecessor marking. We use $[M_0 >$ to denote the set of all markings reachable from the initial marking M_0.

An Example of HPrTNs

The high-level abstraction in Figure 3 (1) shows two super nodes, super place *Philosophers* and super transition *Chopsticks*, which are connected through two arcs with the same label $x + y$ (label constructor + indicates nondeterministic flow relation, i.e., either x, or y, or x and y). We only list (distinct) variable names x and y without using label identifications to simplify our discussion. The low-level refinement in Figure 3, (2) shows the internal structure of *Philosophers* with two states denoted by places *Thinking* and *Eating,* respectively, and the internal structure of *Chopsticks* with two transitions *Take* and *Release* and two places *Avail* and *Used*

denoting the states of chopsticks. The algebraic definition of the net structure is as follows:

P = {*Philosophers*, *Thinking*, *Eating*, *Avail*, *Used*};
T = {*Chopsticks*, *Take*, *Release*};
F = {(*Philosophers*, *Chopsticks*), (*Chopsticks*, *Philosophers*), (*Thinking*, *Thinking*•),
(•*Thinking*, *Thinking*), (*Eating*, *Eating*•), (•*Eating*, *Eating*), (•*Take*, *Take*), (*Take*, *Take*•),
(•*Release*, *Release*), (*Release*, *Release*•), (*Take*, *Used*), (*Avail*, *Take*),
(*Release*, *Avail*), (*Used*, *Release*) };
ρ = { *Philosophers* $\mapsto$ {*Thinking*, *Eating*}, *Chopsticks* $\mapsto$ {*Take*, *Release*, *Avail*, *Used*},
Thinking $\mapsto \varnothing$, *Eating* $\mapsto \varnothing$, *Take* $\mapsto \varnothing$, *Release* $\mapsto \varnothing$, *Avail* $\mapsto \varnothing$, *Used* $\mapsto \varnothing$}

In the underlying specification $SPEC = (S, OP, Eq)$,

(1) S includes elementary sorts such as Integer and Boolean, and also two sorts PHIL and CHOP derived from Integer. S also includes structured sorts such as set and tuple obtained from the Cartesian product of the elementary sorts.
(2) OP includes standard arithmetic and relational operations on Integer, logical connectives on Boolean, set operations, and selection operation on tuples (we use $A[i]$ to denote the ith component of tuple A).
(3) Eq includes equations defining the known properties of the operators.

The net inscription (φ, L, R, M_0) is as follows:

(1) Sorts of places:
$\varphi(Philosophers) = \varphi(Thinking) = \varphi(Eating) = \wp$(PHIL),
$\varphi(Avail) = \wp$(CHOP), $\varphi(Used) = \wp$(CHOP´PHIL).
(2) Label definitions:
Labels are self-evident in this example.
(3) Constraints of transitions:
$R(Take) = (x = u) \wedge (v = x \oplus 1) \wedge (u' = <u, x>) \wedge (v' = <v, x>)$,
$R(Release) = (y = u[2]) \wedge (y = v[2]) \wedge (u' = u[1]) \wedge (v' = v[1])$,
$R(Chopstcks) = True$,
in which $\oplus$ is modulus k addition assuming that there are k philosophers.
(4) The initial marking:
$M_0(Thinking) = \{1, 2, ..., k\}$,
$M_0(Eating) = \{\}$,
$M_0(Philosophers) = M_0(Thinking) \cup M_0(Eating) = \{1, 2, ..., k\}$,
$M_0(Avail) = \{1, 2, ..., k\}$,
$M_0(Used) = \{\}$.

An Approach to Translate a Use Case View into an HPrTN

In the following subsections, we provide an approach to translate a use case view into an HPrTN.

Translating a Use Case Diagram into an HPrTN Net Structure

- **Translating Actors and Associations.** An actor denotes a user of an entity (system, subsystem, or class) and initiates an interaction. Thus, we can use a transition to denote an actor. Each firing of the transition can be viewed as a request for interaction or actor instance. Since an actor is an external entity, the transition is elementary. The tokens generated by the transition depend on the type of requests, which are dependent on specific use cases. Therefore, if an actor connects to more than one use case, we need to introduce an additional transition and its constraint to denote each additional connected use case. The definition provides a dynamic spontaneous view of an actor. Actors with commonalities (i.e., with Generalization and Specialization relationships) are denoted by the same transition and distinguished by the constraint of the transition. The subsequent message exchanges are taken place through free variables in the constraints of transitions. Thus, we have the following translation rules:

Rule 1: Translation rule for a single actor and multiple use cases:

To distinguish individual actor instances and use case instances, tokens generated need to include session identifications and use case identifications, which are defined in net inscription.

Rule 2: Translation rule for multiple actors sharing a common use case:

To distinguish different actors, tokens generated from different actors need to carry actor identifications.

Figure 4. Actor translation rule 1

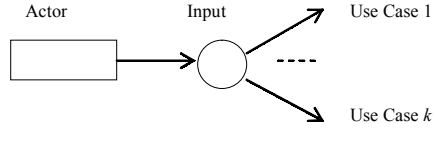


Figure 5. Actor translation rule 2

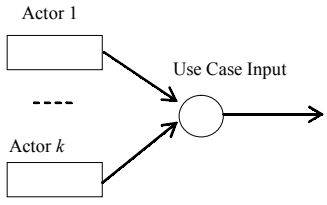
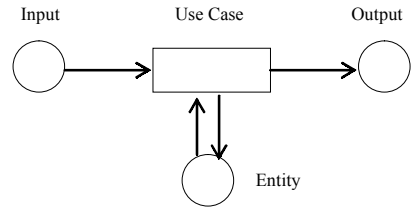


Figure 6. The use case translation rule



- **Translating Use Cases and Associations.** A use case describes the interactions between the users and an entity (a system, a subsystem, or a class) as well as the responses performed by the entity. To define a use case, we need to represent interactions, the entity, and the responses. It is natural to define the interaction pattern by using an input place and a transition pair with proper inscription. The place holds arrival requests and the transition defines behavior changes of the entity. The entity is denoted by another place connected to and from the transition. The entity responses can either be defined implicitly by using the constraint of the transition or explicitly by an output place of the transition. Each firing of the transition indicates a particular interaction or use case instance.

Rule 3: Translation rule for a use case

- **Translating Generalization Relationship.** From the informal description of generalization, it is easy to see that the child use case contains everything in the parent use case and participates in all relationships of the parent use case. The only difference is that a child use case may have some additional behavior sequences. Therefore, we can use a super transition to denote the general (parent) use case and international transitions to denote special (children) use cases and combine the child and the parent use case's constraints to differentiate them.

Rule 4: Translation rule for use case generalization relationship:

The use case description of parent is combined with those of children and translated into constraints associated with the children transitions.

- **Translating <<include>> Relationship.** The include relationship between two use cases is defined by embedding the net structure of the target use case at some location of the net structure of the base use case. The embedding is done depending on the location of the point of inclusion, which is extracted from the use case description. The target case can be inserted before the base case if the target case is invoked first shown in Figure 8 (a); or the target case can be inserted after

Figure 7. Translation rule for generalization

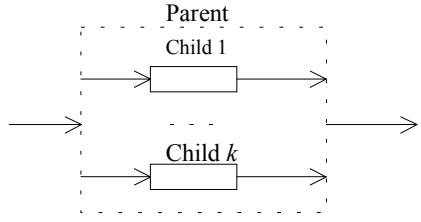


Figure 8 (a). Target case invoked in the beginning of base case

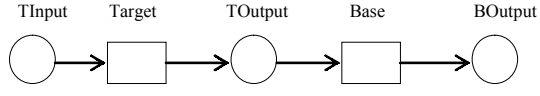


Figure 8 (b). Target case invoked at the end of base case

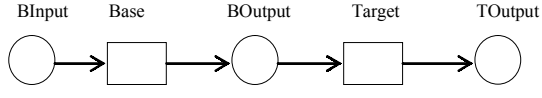
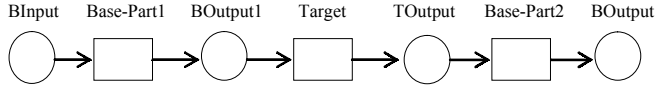


Figure 8 (c). Target case invoked in the middle of base case

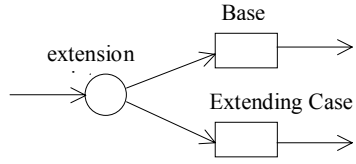


the base case if the target case is invoked at the end of the base case as shown in Figure 8 (b); or the target case is inserted in the middle of the base case, in which a base case is split into two transitions with additional connecting places as shown in Figure 8 (c).

Rule 5: Translation rule for <<include>> relation

- **Translating <<extend>> Relationship.** The informal description of the extend relationship says the relationship contains a condition and a sequence of extension points in the target use case. To define the condition and extension points, it is

Figure 9. Translation rule capturing one extension point



necessary to have a finer view of a use case. Therefore, the target use case is defined by a sequence of the net elements such that a main path defines the essential behavior of the target use case and each branch includes a relevant base use case for extended behavior. It is not difficult to see that an extend relationship is more complicated than an include relationship. In HPrTNs, conditions and thus choices are defined by the following net element and the constraints of the relevant transitions.

Rule 6: Translating rule for <extend>> relation

Translating Use Case Descriptions into an HPrTN Net Inscription

In the previous subsection, we provided an approach to translate a UML use case diagram into a HPrTN net structure. To define the formal semantics of a use case diagram, we need definitions of use cases. In the UML semantic document UML 2003, a variety of options, including plain text, using operations and methods, activity diagrams, state machines, or other behavioral description techniques such as pre- and post-conditions, are listed as possible means for defining the semantics of UML use cases. Most of the mentioned alternatives only provide the static semantics with the exception of state machines. The advantages of using HPrTN net to define UML use case views are that HPrTNs are executable (He, 1996), and there are several behavioral analysis techniques for HPrTNs. Once an HPrTN is derived from a UML use case diagram, we can carry out a variety of analysis, including simple simulation to formal verification to reveal potential defaults in the original use case view. Since UML 2003 does not provide any detailed specification with regard to which technique to use and how to use the chosen technique, we can only provide some general ideas on how to translate these different techniques into HPrTNs based on our prior works. More technical details can be found in the relevant works cited in the descriptions. In general, plain English descriptions of use cases are the most difficult to translate since they are less precise, not well structured, and require more extensive human involvement.

In the following subsections, we discuss the alternative descriptions of use cases and the possible Petri net formalization approaches.

Plain Text Description of Use Cases

When plain English is used to define the use cases, we essentially have to create a formal specification based on the English description. The problem is inherently difficult and there is not a general solution, although there are many different ways and heuristics to deal with the problem. Here we point out two Petri net based formalization approaches. First, we can apply the Petri net formalization technique proposed in He and Yang (1992), which adapts the modern strutcured analysis methods in Youdon (1989). Use cases are thus viewed as event lists. A simple predicate transition net is developed for each use case, and these predicate transition nets are then connected according to the overall use case diagram. Second, an intermediate specification is generated using the modern structured analysis methods. Then, the intermediate specification consisting of a data flow diagram, a data dictionary, and a set of process specifications written in decision tables can be systematically translated into an algebraic Petri net using the approach developed in Kan and He (1996), in which the net structure is derived from the data flow diagram, the sorts (or types) of data elements are derived from the data dictionary, and the transition contraints are derived from the decision tables.

Operation and Method Description of Use Cases

While it is not clear whether the operations and methods mentioned in UML 2003 refer to those defined in class definition or not, one thing is sure: they provide more structures than the plain text descriptions. If the operations and methods are not defined in a class context, then we can certainly use the approaches proposed in dealing with plain text descriptions. If the operations and methods are defined in a class context, we offer the following two possible translation approaches. The general object-orientation technique in HPrTNs (He & Ding, 2001) can be used to define the operations and methods when the operations and methods are not defined in classes represented in a UML class diagram. The more specific technique in translating UML class diagrams into HPrTNs (He, 2000a) can be used to integrate the HPrTN defintions of operations and methods in both UML class diagrams and use case diagrams. The main ideas in both of the approaches are similar: A class is defined by a super place, operations are defined by transitions and their meanings are specified by transition constraints, and attributes are defined by places with appropriate sorts. Based on the correspondences between OO entities and HPrTN elements, various relationships such as associations and inheritance can be realized using additional net structures and inscriptions.

Activity Diagram Description of Use Cases

UML activity diagrams are directly derived from low-level Petri nets, thus an HPrTN net structure can be directly obtained from a given activity diagram. However, an activity diagram only shows simple control information and causal relationships between activities; there is no description in existing UML documentation how to use activity

diagrams to define data processing and transformation in use cases. There is also no discussion what is the relationship between an activity and a use case or even an illustration that shows the use of both of these diagrams in an example in existing UML documentation. Therefore, we believe that the suggestion of using activity diagrams to define use cases is not well-defined and cannot be considered as a meaningful alternative approach at this present time.

State Machine Description of Use Cases

State machines, especially UML statechart diagrams, are a suitable means for defining use cases. Statechart diagrams are a formal method and thus provide precise meanings for use cases. The translation from a statechart diagram to an HPrTN net is quite straightforward. The hierarchical (nested) states in a statechart are translated into super places in an HPrTNs and the concurrent states are translated into independent places in an HPrTN. Each elementary state in a statechart diagram is translated into an elementary place in an HPrTN, and each state transition in a statechart diagram is translated into a transition with proper constraint in an HPrTN such that the transition connects two places representing the corresponding states in the statechart diagram. We have developed techniques to translate UML statechart diagrams into HPrTNs (Dong & He, 2001; and Dong et al., 2003).

The Pre- and Post-Condition Description of Use Cases

If the semantics of a use case is given using pre- and post-conditions, we can easily translate them into first order logic formula and inscribe the formula to the transition. We have shown how to derive transition constraints from decision tables (Kang & He, 1996) in which the condition part is essentially preconditions and the action part defines the post-conditions. We also developed a set of heuristics to develop precondition and post-condition style constraints using *PZ* nets (He, 2001), and the heuristics can be adapted to HPrTNs easily.

Analyzing Behavioral Properties of HPrTNs

Several techniques have been developed for analyzing HPrTNs (He, 1995, 1996, 1998). One obvious choice is the simulation technique based on the dynamic semantics of a predicate transition net after an HPrTN is flattened; this can be used to demonstrate any given scenario and uncover requirement errors as well as specification errors in the use case view if the execution deviates from the expected external behavior.

In this chapter, we also discuss one formal proof technique (He, 1995)—temporal induction technique for analyzing invariant properties of HPrTNs.

System invariant properties are a subclass of general system behavioral properties. A system invariant must be true in every system state, that is, every reachable marking of an HPrTN. The formal definition of invariant properties can be found in Alpern and Schneider (1985). The technique is based on an invariance inference rule from Manna and Pnueli (1992) and combines net structural and behavioral reasoning as well as traditional logical reasoning.

Formalizing Invariant Properties

Let $[M_0 >^{\omega}$ denote the set of all valid marking sequences extracted from all execution sequences of a given HPrTN, and σ be a valid marking sequence and $|\sigma|$ be the length of the sequence, and $\sigma(i)$ be the ith state (marking) in σ. W (a first order logic formula) is an invariant property if and only if the following holds:

$$\forall \sigma \in [M_0 >^{\omega} .(\forall i : 0 \le i \le |\sigma| .(\sigma(i) \models W)),$$

where $\sigma(i)$ W denotes that marking $\sigma(i)$ satisfies W, that is, the evaluation of W under marking $\sigma(i)$ yields true. Thus, a safety property holds in every marking of every valid marking sequence. The formulation can be simplified to the following equivalent version in terms of the set of all reachable markings only:

$$\forall M \in [M_0 > .(M \models W).$$

Proving Invariant Properties

In Manna and Pnueli (1992), several temporal logic-based inference rules for invariance, or safety properties, were given. Among the rules, the following basic invariance rule in its state validity form re-formulated in terms of HPrTNs is essential:

The Basic Invariance Rule

B1. $M_0 \Rightarrow W$
B2. $R(t) : \alpha \wedge W \Rightarrow W'$ for each and occurrence α
$M \models W$ for every $M \in [M_0 >$

In the rule, premise B1 requires that the initial marking M_0 imply property W and premise B2 requires that all transitions preserve W. $R(t)$ is the of t. W' is obtained from W by changing the names of variables to their dashed version. Based on the

premises B1 and B2, we conclude that W is valid or is satisfied under any reachable marking from M_0.

We provide the following induction procedure to apply the inference rule:

- **Step 1:** Prove that the initial marking satisfies a system property formula.
- **Step 2:** Assume that the system property formula holds after k events in a state M.
- **Step 3:** Prove that the system property formula holds after k+1 events in any directly reachable state M' from M.

It is easy to see that Steps 2 and 3 in the temporal induction proof technique fulfill Premise B2 of the invariance rule. Furthermore, we only need to consider the firing of a relevant transition and its constraint with regard to the given property under the guidance of the net structure during Step 3. For example, a system deadlock may occur when a particular predicate has a special marking. To show the system does not have the deadlock, we only need to show that transitions connected to this predicate cannot result in this special marking. Therefore, the proof is in general local in the sense that only a subset of the transitions needs to be considered. In general, logical, net structural, and net behavioral reasoning are needed to prove an invariant property. The temporal induction technique is demonstrated in the next example.

A Translation Example

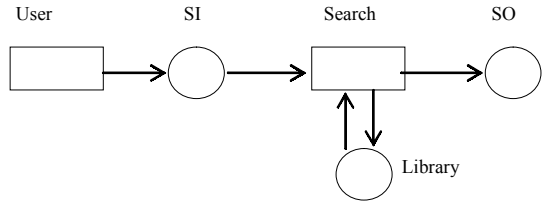
In this section, we first translate the library system described in Figure 2 to an HPrTN structure using the guidelines discussed earlier, and then translate use case descriptions into net inscription according to the guidelines in a previous section to complete the formal definition.

Translating the Use Case Diagram into an HPrTN Structure

Step 1: Translate Use Case Search

Since the actor User initiates the interactions, a transition User is used to generate a token that models an actor instance and hence a use case instance. Applying Rule 1 to actor User and Rule 3 to use case Search, we obtain the following HPrTN structure.

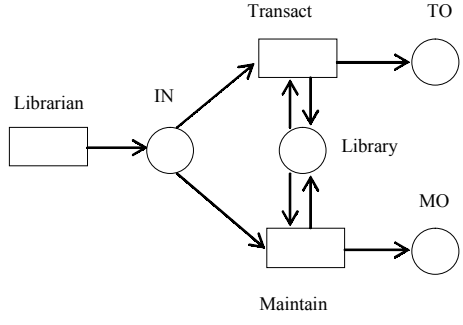
Figure 10 (a). User's search



Step 2: Translate Use Case Transact and Maintenance

Similarly applying Rule 1 to actor Librarian and Rule 3 to use cases Transact and Maintain, we obtain the following HPrTN net structure:

Figure 10 (b). Librarian use cases



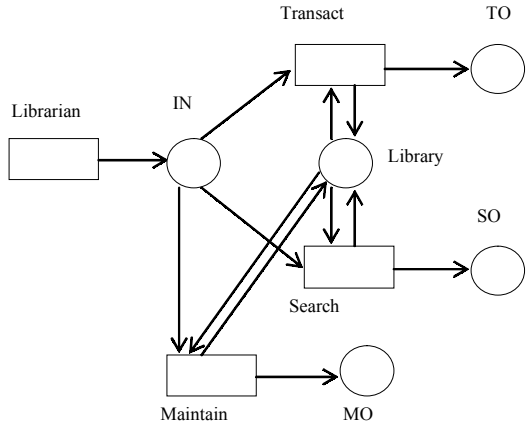
Step 3: Translate the <<extend>> Relation with Regard to Search

An <<extend>> use case defines an optional interaction sequence within the main use case. In this example, place TI serves as a good extension point since no direct behavioral dependence between Transact and Search. After applying Rule 6, we obtain the HPrTN structure illustrated in Figure 10(c).

Step 4: Translate <<include>> Relations with Regard to Validate and Update

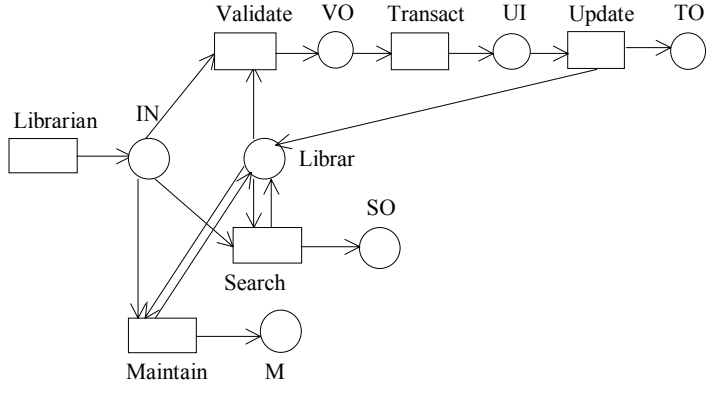
Validation takes place before a borrowing or returning request can be processed, while Update takes place only on proper records. Thus, it is easy to see the pattern in Figure 8 (a) is suitable for use case Validate and the pattern in Figure 8 (b) is suitable for use case

Figure 10 (c). Librarian use cases with <<extend>>



Update. After applying Rule 5 to the <<include>> relationships with regard to Validate and Update, we obtain the following HPrTN structure:

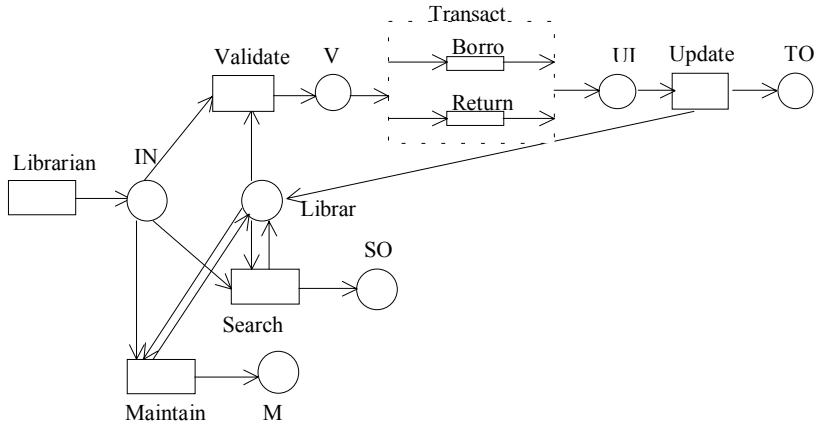
Figure 10 (d). The HPrTN structure with <<include>>



Step 5: Translate the Generalization Relationship with Regard to Transact and Borrow and Return

Applying Rule 4 to the use case Transact in Figure 10 (d), we obtain the following HPrTN structure:

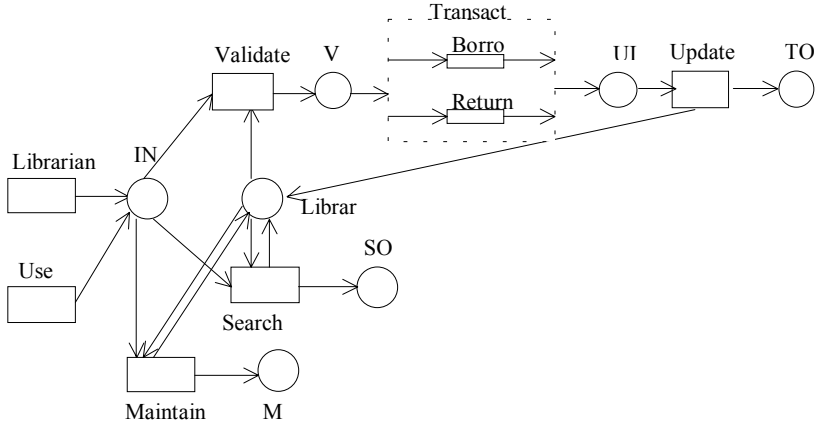
Figure 10 (e). The HPrTN structure with generalization



Step 6: Combine HPrTN structures from multiple actors.

Applying Rule 2 to actors User Figure 10 (a) and Librarian Figure 10 (e), we obtain the following complete HPrTN structure:

Figure 10 (f). The HPrTN structure with generalization



As discussed earlier, different actor instance and use case instances are distinguished through user identifications and session numbers, which are defined in net inscription discussed next.

Translating the Use Case Descriptions into a Net Inscription

Thus far, we have obtained a net structure translated from the use case diagram in Figure 2. In this subsection, we demonstrate how to translate use case descriptions into net inscription. Since the use case descriptions in our example were written in plain English, we use the approach in He and Yang (1992). We omit the translation of use case Maintain and its associated net elements.

Deriving the Sorts of Places

In this section, we define the sort assignment *j* for all places based on the information in the use case descriptions.

(1) Place *IN* – place *IN* holds two types of user requests from users and librarians, respectively. A user request is originated from use case Search and the description mentioned a book title. A librarian request is either from use case Borrow or Return. From the use case descriptions of Borrow or Return, it can be seen that user identification and a book title are needed. To reconcile the two types of requests, a type of ID × BT is needed, where ID denotes the set of valid user identifications and BT denotes the set of all possible book titles. ID also includes a special symbol λ denoting an empty name. Furthermore, to distinguish a different type of request, a request type COM = {**S**, **B**, **R**} is needed. Thus the type of IN is COM × ID × BT.

(2) Place *Library* – the library needs to contain user information. The essential user information needs to contain user identifications ID. The library needs to contain book information and book status. The essential information needs to contain book titles Title and their status, which can be ID indicating their borrowers. If a book title maps to λ, it is available. Thus the type of place *Library* is ID × (BT → ID).

(3) Place *VO* – the type of *VO* combines the request information, library information, and validation results. If we use VAL = {**V**, **I**} to denote a valid user or an invalid user, respectively, then the type of *VO* is (COM × ID × BT) × (ID × (BT → ID)) × VAL.

(4) Place *UI* – the type of *UI* contains the result of Borrow or Return. If we use set OK = {**Y**, **N**} to denote that a Borrow or Return command is successful or not, respectively, place *UI* has the following type (COM × ID × BT) × (ID × (BT → ID)) × VAL × OK.

(5) Place *TO* – place *TO* contains the result of a request and thus has the type (ID × BT) × VAL × OK.

(6) Place *SO* – place *SO* has the type ID × AV, where AV = {**Y**, **N**} indicating the title is in the library or not, respectively.

In summary the sorts of the places are:

$\varphi(IN) = \text{COM} \times \text{ID} \times \text{BT}$,
$\varphi(Library) = \text{ID} \times (\text{BT} \rightarrow \text{ID})$,
$\varphi(VO) = (\text{COM} \times \text{ID} \times \text{BT}) \times (\text{ID} \times (\text{BT} \rightarrow \text{ID})) \times \text{VAL}$,
$\varphi(UI) = (\text{COM} \times \text{ID} \times \text{BT}) \times (\text{ID} \times (\text{BT} \rightarrow \text{ID})) \times \text{VAL} \times \text{OK}$,
$\varphi(TO) = (\text{ID} \times \text{BT}) \times \text{VAL} \times \text{OK}$,
$\varphi(SO) = \text{ID} \times \text{AV}$.

Deriving Arc Label Definitions

We use the following convention in defining arc labels: A lower case variable x denotes an individual variable / or a singleton set, and an upper case W denotes a set variable. The arc labels are derived from the sorts of connecting places. Based on the types of the places, we have the following arc label definitions:

$L(Librarian, IN) = L(User, IN) = x$, $\quad L(IN, Validate) = L(IN, Search) = x$,
$L(Library, Validate) = W$,
$L(Validate, VO) = L(VO, Borrow) = L(VO, Return) = \langle x, W, y \rangle$,
$L(Borrow, UI) = L(Return, UI) = L(UI, Update) = \langle x, W, y, z \rangle$,
$L(Update, TO) = \langle x', y, z \rangle$,
$L(Update, Library) = W'$,
$L(Search, Library) = L(Library, Search) = W$,
$L(Search, SO) = \langle x', y \rangle$.

Deriving Constraint Definitions

We use $x[k]$ to denote the project operation of kth component of variable x.

The constraint of *Librarian* only has a post-condition defined as follows:

$$R(Librarian) = (x[1] \in \{\mathbf{B}, \mathbf{R}\} \wedge x[2] \in \text{ID} \wedge x[3] \in \text{BT})$$

The firing of transition *Librarian* generates a token x with a command type of either **B** or **R**, a user identification, and a book title.

The constraint of *User* only has a post-condition defined as follows:

$$R(User) = (x[1] = \mathbf{S} \wedge x[2] = \mathbf{S} \wedge x[3] \in \text{BT})$$

The constraint of *Validate* is defined as follows:

$$R(Validate) = (x[1] \in \{\mathbf{B}, \mathbf{R}\}) \wedge (x[2] \in W[1] \Rightarrow y = \text{`}\mathbf{V}\text{'} \vee x[2] \notin W[1] \Rightarrow y = \text{`}\mathbf{I}\text{'}),$$

The validation of user request checks whether a provided user identification $x[2]$ is in the set of valid library users $W[1]$.

The constraint of *Search* is defined as follows:

$$R(Search) = (x[1] \in \mathbf{S}) \wedge (x[3] \in \mathbf{dom}\ W[2] \Rightarrow y = \text{`}\mathbf{Y}\text{'} \vee x[3] \notin \mathbf{dom}\ W[2] \Rightarrow y = \text{`}\mathbf{N}\text{'}) \wedge (x' = x[2]),$$

The constraint of *Borrow* is defined as follows:

$$R(Borrow) = (x[1] = \mathbf{B}) \wedge (x[3] \in \mathbf{dom}\ W[2] \Rightarrow (W[2](x[3]) \neq \lambda \Rightarrow z = \text{`}\mathbf{N}\text{'} \vee W[2](x[3]) = \lambda \Rightarrow z = \text{`}\mathbf{Y}\text{'}) \vee x[3] \notin \mathbf{dom}\ W[2] \Rightarrow z = \text{`}\mathbf{N}\text{'}),$$

The precondition of *Borrow* specifies whether the requested book title is available or not. Based on the precondition, a '**N**' or '**Y**' is output as a part of post-condition.

The constraint of *Return* is defined as follows:

$$R(Return) = (x[1] = \mathbf{R}) \wedge (x[3] \in \mathbf{dom}\ W[2] \Rightarrow (W[2](x[3]) = x[2] \Rightarrow z = \text{`}\mathbf{Y}\text{'} \vee W[2](x[3]) \neq x[2] \Rightarrow z = \text{`}\mathbf{N}\text{'}) \vee x[3] \notin \mathbf{dom}\ W[2] \Rightarrow z = \text{`}\mathbf{N}\text{'}),$$

The constraint of *Return* is defined similarly.

The constraint of *Update* is defined as follows:

$$R(Update) = ((y = \text{`}\mathbf{V}\text{'}) \wedge (z = \text{`}\mathbf{Y}\text{'}) \Rightarrow (x[1] = \mathbf{B} \Rightarrow W'[1] = W[1] \wedge W'[2] = W[2] - \{x[3] \mapsto \lambda\} \cup \{x[3] \mapsto x[2]\}) \vee (x[1] = \mathbf{R} \Rightarrow W'[1] = W[1] \wedge W'[2] = W[2] - \{x[3] \mapsto x[2]\} \cup \{x[3] \mapsto \lambda\})) \vee ((y \neq \text{`}\mathbf{V}\text{'}) \vee (z \neq \text{`}\mathbf{Y}\text{'}) \Rightarrow W' = W).$$

The *Update* is unsuccessful if a user is not a valid user $y \neq$ '**V**' or the transaction condition is not met $z \neq$ '**Y**', as a result the library content is unchanged $W' = W$. The *Update* is successful if both of the conditions are satisfied, and the book status is changed.

Deriving the Initial Marking

Initially, there are m users in ID and n books in BT such that all the books are available in place *Library*.

Analyzing an Invariant Property

We demonstrate the application of the structured induction analysis technique through analyzing one invariant property of the use case specification in Figure 10 (f). The invariant property specifies that all copies in the library must be either available for checkout or be checked out.

The property can be formulated as follows:

$$\forall W \in Library.(\forall co \in W[2].(co[2] = \lambda \vee co[2] \in W[1])) \qquad (*)$$

The formula states that any copy *co*, which is a maplet consisting of a book title and a user name, is either available denoted by λ, or checked out with a valid user name in $W[1]$.

We apply the temporal induction procedure to prove (*). In the following proof outline, we skip explanations of simple logical reasoning.

Proof outline of formula (*):

- **Step 1:** Under the initial marking $M0$, (*) is true since all the books are available for checkout.
- **Step 2:** Assume (*) holds after k transactions in a marking M:

  $$\forall co \in W[2].(co[2] = \lambda \vee co[2] \in W[1])$$

- **Step 3:** Prove (*) holds after k+1 transactions in a state M' such that $M[t/\alpha>M'$ for a transition t with an occurrence mode α. We examine each transition firing in turn:

Case 1: Firing transitions *Librarian*, *User*, *Validate*, and *Borrow*/*Return*:

A firing of any transition listed has no effect on place *Library*, thus (*) holds from the induction assumption;

Case 2: Firing transition *Search*:

A firing of transition *Search* does not change *Library* since the labels from and to *Library* are both W. Thus (*) holds from the induction assumption.

Case 3: Firing transition *Update*:

(1) From the disjunct $((y \neq \text{`}\mathbf{V}\text{'}) \vee (z \neq \text{`}\mathbf{Y}\text{'}) \Rightarrow W' = W)$ of the constraint R(Update): *Library* content is unchanged, thus (*) is true from the induction assumption;

(2) From the disjunct $((y = \text{`}\mathbf{V}\text{'}) \wedge (z = \text{`}\mathbf{Y}\text{'}) \Rightarrow$

$(x[1] = \mathbf{B} \Rightarrow W'[1] = W[1] \wedge W'[2] = W[2] - \{x[3] \mapsto \lambda\} \cup \{ x[3] \mapsto x[2]\}) \vee$

$(x[1] = \mathbf{R} \Rightarrow W'[1] = W[1] \wedge W'[2] = W[2] - \{ x[3] \mapsto x[2]\} \cup \{x[3] \mapsto \lambda\}))$

W'[2] is changed either from available to checked out or from checked out to available. Thus (*) holds from the induction assumption.

Therefore, (*) holds after firing transition *Update* from (1) to (5).

Since all possible transition firings from a given marking *M* have been considered and all maintain the property (*), (*) is thus proved by the temporal induction procedure.

Related Work

In Back et al. (1999), the refinement calculus (Back & Wright, 1998) was used to define the formal semantics of use cases. Given a use case diagram, a supplemental formal semantic document, called a contract, was written in refinement calculus. The contract was further used to analyze the behavior of the use case diagram. Their work was very much like the net inscription definition part in our approach. However, they did not deal with the translation of a complete use case diagram, and they did not consider how to formally define use case relationships such as <<extend>>, <<include>>, and generalization. Our approach provides a translation technique to map the syntax of a use case diagram into an HPrTN structure.

In Overgaard and Palmkvist (1998), the formalization of the structure of UML use cases in terms of the sequences of actions defined in predicate logic was discussed, but no analysis was shown.

In Lee, Cha, and Kwon (1998), a modular low-level Petri net model, called constraints-based modular Petri nets (CMPNs), was used to define use cases. A translation technique was provided to map a set of use cases to a CMPN net structure and the Petri net simulation method was used to detect potential incompleteness and inconsistency in the given use cases. Their technique was developed to handle general use cases in requirements engineering but not aimed at the UML use case diagrams. Furthermore, their Petri net model does not support data definition and analysis, which are necessary in UML use case diagrams.

Conclusion

In this chapter, an approach to define and analyze use case view in UML using HPrTNs is presented. We have shown how to define the structure and semantics of a use case view in terms of HPrTN elements and provided a behavioral property analysis technique. We have demonstrated our approach through an example. A correctly defined case use model establishes the foundation for subsequent system development and furthermore provides a sound basis for future system evolution.

Compared with UML notations, HPrTNs are relatively more difficult to understand and use. HPrTNs are a formal method having a formal semantics and supporting formal analysis, and thus are intrinsically more complicated than informal methods. Furthermore, HPrTNs define all system aspects (functionality, data, structure, and behavior) in one notation while UML offers different notations such as Class diagrams, Interaction diagrams, State diagrams, and Activity diagrams for different system aspects. Although this multiple aspects definition may be more complex, it has the advantage of providing a unified semantic domain facilitating cross view consistency checking.

Our purpose of formalizing UML diagrams is not to substitute UML diagrams using HPrTNs, rather we want to use HPrTNs to supplement UML diagrams for formal analysis. Based on our extensive experience on research of formal methods as well as those of other researchers, it is evident that it is extremely difficult, if not possible, to train software engineers to directly use formal methods. A wise and feasible approach is to let software engineers use informal or semi-formal notations such as UML and have special trained analysts to analyze these informal specifications and design through a translation to some formal model. Based on our prior work on using HPrTNs for formalizing UML notations, we see the following potential benefits of our approach to define use case view using HPrTNs. First, our translation approach provides a systematic way to integrate multiple use cases based on given use case diagrams and thus supports incremental and scalable translation. Second, we believe that HPrTNs can serve as a unified notation for other UML graphical notations, that is, class diagrams, object diagrams, sequence diagrams, collaboration diagrams, statechart diagrams, and activity diagrams. We have developed an approach to formally define UML class diagrams using HPrTNs (He, 2000a), and an approach in formalizing statechart diagrams and collaboration diagrams (Dong & He, 2001; Dong et al., 2003). By translating an individual UML diagram into an HPrTN, we can formally analyze its structure and behavior, and thus reveal possible modeling and design errors in the original UML diagrams. Third, we believe that a single HPrTN can provide a unified model to define many different views provided by separate diagrams in UML. Building a unified HPrTN for multiple views defined in several UML diagrams can help detect inconsistencies among these UML models. We are exploring ways to facilitate the practical use of our translation techniques, including developing techniques to map detected errors to relevant use cases or relationships in the original use case view and building prototype tools to assist the translation process.

Acknowledgments

We thank the two reviewers for helping us to improve the presentation of this work. This research was supported in part by the Office of Naval Research of the USA under grant N00014-98-1-0591, by the National Science Foundation of the USA under grant HRD-0317692, and by the National Aeronautics and Space Administration of the USA under grant NAG2-1440.

References

Alpern, B., & Schneider, F. (1985). Defining liveness. *Information Processing Letters*, *21*(4), 181-185.

Back, R. & Wright, J. (1998). *Refinement calculus: A systematic introduction*. Berlin, Germany: Springer-Verlag.

Back, R., Petre, L., & Paltor, I. (1999). Analyzing UML use cases as contracts. *Lecture Notes in Computer Science*, *1723,* 518-533.

Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The unified modeling language user guide*. Reading, MA: Addison Wesley Longman.

Dong, Z., & He, X. (2001). Integrating UML statechart and collaboration diagrams using hierarchical predicate transition nets. *Lecture Notes in Informatics*, *P-7,* 99-112.

Dong, Z., Fu, Y., & He, X. (2003). Deriving hierarchical predicate/transition nets from statechart diagrams – A case study. *Proceedings of the SEKE 2003*, California (pp. 150-157).

Evans, A. (1998). Reasoning with UML class diagrams. *Proceedings of the Second IEEE Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, (pp. 102-113).

France, R., Evans, A., Lano, K., & Rumpe, B. (1998). Developing the UML as a formal modeling notation. *Computer Standards and Interfaces*, *19,* 325-334.

Genrich, H. (1987). Predicate transition nets. *Lecture Notes in Computer Science*, *254,* 205-247.

Harel, D. (1988). On visual formalisms. *Communications of the ACM*, *31,* 514-530.

He, X. (1995). A method for analyzing properties of hierarchical predicate transition nets. *Proceedings of the 19th International Computer Software and Applications Conference*, Dallas, (pp. 50-55).

He, X. (1996). A formal definition of hierarchical predicate transition nets. *Lecture Notes in Computer Science, 1091*, 212-229.

He, X. (1998). Transformations on hierarchical predicate transition nets: Abstractions and refinements. *Proceedings of the 22nd International Computer Software and Application Conference* (*COMPSAC'98),* Vienna, Austria, (pp. 164-169).

He, X. (2000a). Formalizing UML class diagrams – A hierarchical predicate transition net approach. *Proceedings of the 24th International Computer Software and Application Conference* (*COMPSAC 2000),* Taiwan, (pp. 217-222).

He, X. (2000b, August). Formalizing use case diagrams in hierarchical predicate transition nets. *Proceedings of the IFIP 16th World Computer Congress*, Beijing, China, (pp. 484-491).

He, X. (2000c). Translating hierarchical predicate transition nets into CC++ programs. *Information and Software Technology*, *42*(7), 475-488.

He, X. (2001). PZ nets – A formal method integrating Petri nets and Z. *Information and Software Technology*, *41*(1), 1-18.

He, X. & Ding, Y. (2001). Object-orientation in hierarchical predicate transition nets. *Lecture Notes in Computer Science*, *2001,* 196-215.

He, X. & Lee, J.A.N. (1991). A methodology for constructing predicate transition net specifications. *Software - Practice and Experience*, *21*(8), 845-875.

He, X. & Yang, C.H. (1992). Structured analysis using hierarchical predicate transition nets. *Proceedings of 16th International Computer Software and Applications Conference*, Chicago, (pp. 212-217).

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The unified modeling language development process.* Reading, MA: Addison Wesley Longman.

Jacobson, I. Christerson, M., Jonsson, P., & Overgaard, G. (1992). *Object-oriented software engineering: A use case driven approach*. Reading, MA: Addison-Wesley.

Kan, C. & He, X. (1996). A method for constructing algebraic Petri nets. *Journal of Systems and Software*, *35,* 12-27.

Kim, S. & Carrington, D. (1999). Formalizing the UML class diagram using object-Z. *Proceedings of UML'99*, *Lecture Notes in Computer Science* (Vol. 1723, pp. 83-98).

Knapp, A. (1999). A formal semantics for UML interactions. *Proceedings of UML'99, Lecture Notes in Computer Science* (Vol. 1723, pp. 116-130).

Lano, K. & Bicarregui, J. (1998). Formalizing the UML in structured temporal theories. *Proceedings of the Second ECOOP Workshop on Precise Behavioral Semantics*, (pp. 105-121). Springer-Verlag.

Lee, W., Cha, S., & Kwon, Y. (1998). Integration and analysis of use cases using modular Petri nets in requirements engineering. *IEEE Trans. on Software Engineering*, *24*(12), 1115-1130.

Manna, Z. & Pnueli, A. (1992). *The temporal logic of reactive and concurrent systems – Specification*. Springer-Verlag.

McUmber, W. & Cheng, B. (1999). UML-based analysis of embedded systems using a mapping to VHDL. *Proceedings of IEEE High Assurance Software Engineering*, Washington, DC, November.

McUmber, W. & Cheng, B. (2001, May). A generic framework for formalizing UML. *Proceedings of IEEE International Conference on Software Engineering* (*ICSE01),* Toronto, Canada.

Overgaard, G. & Palmkvist, K. (1998). A formal approach to use cases and their relationships. *Lecture Notes in Computer Science*, *1618,* 309-317.

Rumbaugh, J., Booch, G., & Jacobson, I. (1999). *The unified modeling language reference manual.* Reading, MA: Addison Wesley Longman.

Saldhana, S. & Shatz, S. (2000). UML diagrams to object Petri net models: an approach for modeling and analysis. *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering*, Chicago, (pp. 103-110).

Saldhana, S., Shatz, S., & Hu, Z. (2001). Formalization of object behavior and interactions from UML models. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE), 11*(6), 643-673.

Shroff, M. & France, R. (1997). Towards a formalization of UML class structures in Z. *Proceedings of COMPSAC'97*, Washington, DC.

UML (2003). *Unified Modeling Language*. Version 1.5. Available online at *http://www.omg.org*

Yourdon, E. (1989). *Modern structured analysis*. Englewood Cliffs, NJ: Prentice Hall.

Chapter IX

Formal Specification of Software Model Evolution Using Contracts

Claudia Pons, Universidad Nacional de La Plata, Argentina

Gabriel Baum, Universidad Nacional de La Plata, Argentina

Abstract

During the object-oriented software development process, a variety of models of the system is built. All these models are semantically overlapping and together represent the system as a whole. In this chapter, we present a classification of relationships between models along three different dimensions, proposing a formal description of them in terms of mathematical contracts, where the software development process is seen as involving a number of agents (the development team and the software artifacts) carrying out actions with the goal of building a software system that meets the user requirements. In this way, contracts can be used to reason about correctness of the development process, and to compare the capabilities of various groupings of agents in order to accomplish a particular contract. The goal of the proposed formalization is to provide formal foundations for tools that perform intelligent analysis on models assisting software engineers through the software life cycle.

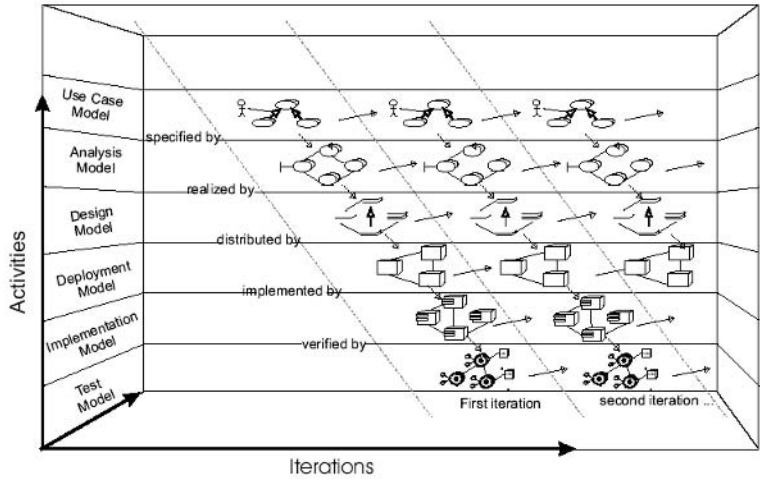
Introduction

A software development process is a set of activities that jointly convert users' needs to a software system. Modern software development processes, such as the Unified Process (Jacobson, Booch, & Rumbaugh, 1999), are iterative and incremental, they repeat over a series of iterations making up the life cycle of a system. Each iteration takes place over time and consists of one pass through the requirements, analysis, design, implementation, and test activities, building a number of different models. Due to the incremental nature of the process, each iteration results in an increment of models built in previous iterations. This creates a natural relationship between the elements among different phases and iterations; elements in one model can be related to elements in another model. For instance, a use case (in the use case model) can be traced to a collaboration (in the analysis or design model) representing its realization. Figure 1 lists the classical phases or activities — requirements, analysis, design, implementation, and test — in the vertical axis and the iteration in the horizontal axis. Three different dimensions are distinguished in order to classify relationships between models:

- horizontal dimension (internal dimension)
- vertical dimension (activity dimension)
- evolution dimension (iteration dimension)

The horizontal dimension deals with relations between submodels that coexist consistently making up a more complex model. The UML incorporates several sublanguages, each one allowing a specific view on the system. Models of different viewpoints have a certain overlap, for instance, an analysis model consists of sequence diagrams and

Figure 1. Dimensions in the development process



collaboration diagrams both representing different aspects of the behavior of the system.

The vertical dimension considers relations between models belonging to the same iteration in different activities (e.g., a design model realizing an analysis model). Two related models represent the same information but at different levels of abstraction.

The evolution dimension considers relations between artifacts belonging to the same activity in different iterations (e.g., a use case is extended by another use case). In this dimension, new models are built or derived from previous models by adding new information that was not considered before or by modifying or detailing previous information.

An essential element to the success of the software development process is the support offered by case tools. Existing case tools facilitate the construction and manipulation of models, but in general, they do not provide checks of consistency between models along either vertical or evolution dimension. Tools neither provide automated evolution of models (i.e., propagation of changes when a model evolves, to its dependent models). The weakness of tools is mainly due to the lack of a general underlying formal foundation for the software development process (particularly focused on relations between models).

To overcome this problem, we propose to apply the well-known mathematical concept of contract to the specification of software development processes by introducing the concept of software process contract (sp-contract). Sp-contracts introduce precision of specification, avoiding ambiguities and inconsistencies, and enabling developers to reason about the correctness of their joint activities. The goal of the proposed formalism is to provide foundations for case tools assisting software engineers during the development process. Sp-contracts provide a formalization of software artifacts and their relationships. They clearly specify pre- and post-conditions for each software development task, allowing for the verification of consistency between models through evolution.

The remainder of this chapter is organized as follows. First, we describe the underlying formalism we use to develop our proposal. Then, we introduce the concept of software process contract (sp-contract), which constitutes our proposal to improve formality of software development process. The next section contains some ideas about future trends and the construction of a case tool based on sp-contracts. Finally, we present the conclusion and related works.

Background: Notion of Software Contract

Generally, a computation can be seen as involving a number of agents (objects) carrying out actions according to a document (specification, program) that has been laid out in advance. This document represents a contract between the agents involved. A contract imposes mutual obligations and benefits; it protects both sides (the client and the contractor):

- It protects the client by specifying how much should be done; the client is entitled to receive a certain result.
- It protects the contractor by specifying how little is acceptable; the contractor must not be liable for failing to carry out tasks outside of the specified scope.

The notion of contract regulating the behavior of a software system has been investigated by several authors (Andrade & Fiadeiro, 1999; Back, Petre, & Porres Paltor, 1999; Helm, Holland, & Gangopadhyay, 1990; Meyer, 1992; Meyer, 1997). In particular, in our work, we apply the formalism of contracts proposed by Ralf Back which is based on the Refinement Calculus (Back & von Wright, 1998).

The refinement calculus is a logical framework for reasoning about programs. It is concerned with two main questions: Is a program correct with respect to a given specification? And, how can we improve, or refine, a program while preserving its correctness? Both programs and specifications can be seen as special cases of a more general notion, that of a contract between independent agents. Refinement is defined as an ordering relation between contracts. Correctness is a special case of refinement where a specification is refined by a program.

The refinement calculus is formalized within higher order logic, allowing us to prove the correctness of contracts and to calculate contracts refinements in a rigorous, mathematically precise manner. The refinement calculus is equipped with automatic tools: the Mechanised Reasoning Group led by Joakim von Wright has developed a system for supporting program derivation, precondition calculation and correctness calculation within the Refinement Calculus framework that is called the Refinement Calculator (Butler, Grundy, Langbacka, Ruksenas, & Von Wright, 1997; Celiku & von Right, 2002). This system is based on the HOL theorem prover.

Contract Language

Consider a collection of agents, where each agent has the capability to change the world in various ways through its actions and can choose different courses of action. The behavior of agents and their cooperation is regulated by contracts. The contract can stipulate that the agent must carry out actions in a specific order. This is written as a sequential statement $S_1;...;S_m$, where $S_1,...,S_m$ are the individual actions that the agent has to carry out. A contract may also require the agent to choose one of the alternative actions S1,...Sm. The choice is written as the alternative statement S1È...ÈSm.

The world is described as a state s. The state space S is the set of all possible states s. The state is observed as a collection of attributes x_1, x_2, ...,x_n, each of which can be observed and changed independently of the others. An agent changes the state by applying a function f to the present state s, yielding a new state f.s. These functions mapping states to states are the most primitive form of action that agents can carry out. An example of state transformer is the assignment x:=exp, that updates the value of attribute x to the value of the expression exp.

The language for contracts is simple:

$$S ::= \langle f \rangle \mid \textbf{if } p \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \mid S_1 \textbf{ ; } S_2 \mid \textbf{assert}_a\ p \mid \textbf{R}_a \mid S_1 \cup_a S_2 \mid \textbf{rec } X \bullet S$$

Here a stands for an agent while f stands for a state transformer, p for a state predicate (i.e., a boolean function p:$\Sigma \rightarrow$Bool) and R for a state relation (i.e., relation R:$\Sigma \rightarrow \Sigma \rightarrow$ Bool relates a state σ to a state σ' whenever R.σ.σ' holds).

Each statement in this language describes a contract for an agent. Intuitively, a contract is executed as follows:

The functional update <f> changes the state according to the state transformer f, that is, if the initial state is σ_0 then the final state is f.σ_0. An assignment statement is a special kind of update where the state transformer is expressed as an assignment. For example, the assignment statement <x:=x+y> requires the agent to set the value of attribute x to the sum of the values of attributes x and y.

In the conditional composition if p then S_1 else S_2 fi, S_1 is carried out if p holds in the initial state, and S_2 otherwise. In the sequential composition S_1; S_2, statement S_1 is carried out first, followed by S_2.

An assertion $assert_a$ p , for example, $assert_a$ (x+y=0) expresses that the sum of (the values of) x and y in the state must be zero. If the assertion holds at the indicated place when the agent *a* carries out the contract, then the state is unchanged, and the rest of the contract is carried out. If, on the other hand, the assertion does not hold, then the agent has breached the contract.

The relational update and choice both introduce nondeterminism into the language of contracts. Both are indexed by an agent which is responsible for deciding how the nondeterminism is resolved. The relational update Ra requires the agent a to choose a final state σ' so that R.σ.σ' is satisfied, where σ is the initial state. In practice, the relation is expressed as a relational assignment. For example, updatea {x := x′| x′ <x} expresses that the agent *a* is required to decrease the value of the program variable x. If it is impossible for the agent to satisfy this, then the agent has breached the contract.

The statement $S_1 \cup_a S_2$ allows agent *a* to choose which is to be carried out, S_1 or S_2.

Finally, recursive contract statements are allowed. A recursive contract is defined using an equation of the form X = S, where S may contain occurrences of the contract variable X. With this definition, the contract X is intuitively interpreted as the contract statement S, but with each occurrence of statement variable X in S treated as a recursive invocation of the whole contract S. It also is permitted the syntax (rec X•S) for the contract X defined by the equation X=S. An important special case of recursion is the while-loop which is defined in the usual way: while p do S od =(rec X•if p then S ; X else skip fi) where skip is the well-known “do nothing” statement.

Cooperation Contract

Consider a set of agents that work on the same state independently of each other. Each agent has a will of its own and makes decisions for itself. If these agents want to cooperate, they need a contract that stipulates their respective obligations.

A typical situation is that one of the agents acts as a server, and the other as clients. Assume that a client follows contract S,

Contract S = (f1 $\cup$ skip) ; T ; f2

Where f1 and f2 are primitive actions and T is the contract for the server,

Contract T = f3 $\cup$ f4

The occurrence of T in the contract statement S signals that the client asks the server to carry out its contract T.

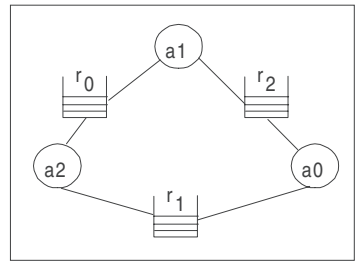
We can combine the two statements S and T into a single contract statement regulating the behavior of both agents. The combined contract is described by

Contract V = (f1 $\cup_{client}$ skip) ; (f3 $\cup_{server}$ f4) ; f2

The combined collaborative contract is the result of substituting the contract statement T for the invocation on T in the contract S and explicitly indicating for each choice which agent is responsible for it.

Another form of interaction between agents occurs when they need to synchronize their individual actions. Assume that the agents (a_0, a_1, a_2) are placed in a ring, with a collection of resources situated between them (r_i is the collection of resources placed between agents a_{i-1} and a_{i+1}, where modulo-3 arithmetic is used). This situation is illustrated in Figure 2.

Figure 2. Resource game



Each agent a_i has access to the resource in r_{i-1} and r_{i+1} but not to r_i. Resources are nonrenewable, and we assume that the agents take turns grabbing one of them from either side (left or right). An agent also can choose to do nothing. Initially, no agents have grabbed any resource, and the resources are evenly distributed,

$init = n_0,n_1,n_2,r_0,r_1,r_2 := 0,0,0,m,m,m$

where n_i models the number of resources that agent a_i has grabbed and m is the initial number of resources at each point.

The alternatives open to agent a_i are described by the following contract:

Contract $S_i = grabl_i \cup_{ai} skip \cup_{ai} grabr_i$

Where

$grabl_i = assert_{ai}\ r_{i-1} > 0\ ;\ n_i,\ r_{i-1} := n_i + 1,\ r_{i-1} - 1$

$grabr_i = assert_{ai}\ r_{i+1} > 0\ ;\ n_i,\ r_{i+1} := n_i + 1,\ r_{i+1} - 1$

Now the whole system can be described as the combination of subcontract S_0, S_1 and S_2 into a single contract statement that regulates the behavior of the three agents, as follows:

Contract System = init; while $r_o + r_1 + r_2 > 0$ do S_0 ; S_1 ; S_2 od

According to this contract, on every round the order of choices is deterministic: agent a_0 chooses first, then a_1 and finally a_2. It is possible to write a different contract permitting a different order of choices.

Semantics of Contracts: The Rules of a Game

Agents try to achieve their goals, that is, to reach a new, more desirable state. The desired states are described by giving condition that they have to satisfy (the post-condition). The possibility of an agent to achieve such a desired state depends on the functions that it can use to change the state.

Given a contract for a single agent and a desired post-condition, we can ask whether the agent following the contract can establish the post-condition. This will depend on the initial state, the state in which the agent starts to carry out the contract. For instance, consider the contract

Contract S = x:=x+1 ∪ x:=x+2

The agent can establish the post-condition x=2 if x=1 or x=0 initially. When x=1 initially, the agent should choose the first alternative; but when x=0, the agent should choose the

second alternative. But the agent cannot achieve its goal from any initial state satisfying either $x>1$ or $x<0$.

On the other hand, an agent cannot be required to follow a contract if the assumptions that it makes are violated for non-allied agents. Violating the assumptions releases the agent from the contract.

The main concern with a contract is to determine whether a set of agents, say A, can use the contract to achieve the stated goals. In this sense, agents have to make the right choices in their cooperation with other agents, which are pursuing different goals that need not be in agreement with their goals. The other agents need not to be hostile; they just have different priorities and are free to make their own choices. However, because no one agent in A can influence the other agents in any way, they have to be prepared for the worst and consider the other agent as hostile. From the point of view of a specific agent or a group of agents, it is therefore interesting to know what outcomes are possible regardless of how the other agents resolve their choices.

As far as analyzing what can be achieved with a contract, it is justified to consider the agents involved as the opponents in a game. The actions that the agents can take are the moves in the game. The rules of the game are expressed by the contract; it states what moves the opponents can take and when.

A player in the game is said to have a winning strategy in a certain initial state if the player can win (by doing the right moves) no matter what the opponents do.

Consider the situation where the initial state σ is given and a group of agents A agree that their common goal is to use contract S to reach a final state satisfying q. Satisfaction of a contract (denoted by $\sigma\{S\}q$) corresponds to the existence of a winning strategy. It means that $\sigma\{S_A\}q$ holds if and only if the set of agents has a winning strategy to reach the goal q when playing with the rules S, when the initial state of the game is σ.

If some of the agents in A are forced to breach an assertion, then the coalition loses the game. If the opponents are forced to breach an assertion, they lose the game, and the coalition wins. In this way, an agent can win the game either by reaching a final state that satisfies the post-condition or by forcing the opponents to breach an assertion.

This notion of satisfaction is precisely defined, in the following way: The predicate transformer $\mathbf{wp}_A.S$ maps post-condition q to the set of all initial states σ from which the agents in A jointly have a winning strategy to reach the goal q. Thus, $\mathbf{wp}_A.S.q$ is the weakest precondition that guarantees that the agents in A can cooperate to achieve post-condition q. This means that a contract S for a coalition A is mathematically seen as an element (denoted by $\mathbf{wp}_A.S$) of the domain $\mathbb{P}\Sigma \rightarrow \mathbb{P}\Sigma$. Then, the satisfaction of contracts is captured naturally by the notion of weakest precondition, as follows: $\sigma\{S\}q \equiv \mathbf{wp}_A.S.q.\sigma$

The definition of the predicate transformer is as follows. See Back & von Wright (1998) for a more detailed explanation:

(i) $\mathbf{wp}_A.\langle f\rangle.q = (\lambda\sigma.q.(f.\sigma))$

(ii) $\mathbf{wp}_A.(\text{if p then } S_1 \text{ else } S_2 \text{ fi}).q = (p \cap \mathbf{wp}_A.S_1.q) \cup (\neg p \cap \mathbf{wp}_A.S_2.q)$

(iii) $\mathbf{wp}_A.(S_1;S_2).q = \mathbf{wp}_A.S_1.(\mathbf{wp}_A.S_2.q)$

(iv) $\mathbf{wp}_A.(\text{assert}_a\ p).q = \lambda\sigma.(p.\sigma \wedge q.\sigma)$, if $a \in A$

$\lambda\sigma.(\neg p.\sigma \vee q.\sigma)$, if $a \notin A$

(v) $\mathbf{wp}_A.R_a.q = \lambda\sigma.\exists\sigma' \bullet R.\sigma.\sigma' \wedge q.\sigma'$, if $a \in A$

$\lambda\sigma.\forall\sigma' \bullet R.\sigma.\sigma' \rightarrow q.\sigma'$, if $a \notin A$

(vi) $\mathbf{wp}_A.(S_1 \cup_a S_2).q = \mathbf{wp}_A.S_1.q \cup \mathbf{wp}_A.S_2.q$, if $a \in A$

$\mathbf{wp}_A.S_1.q \cap \mathbf{wp}_A.S_2.q$, if $a \notin A$

Notion of a Software Process Contract

While the notion of a formal contract regulating the behavior of software agents is accepted, the concept of contract regulating the activities of software developers is quite vague. In general, there is no explicit contract establishing obligations and benefits of members of the development team. At best, the development process is specified by either a graph of tasks or object-oriented diagrams in a semi-formal style, while in most cases activities are carried out on-demand, with little previous planning.

However, a disciplined software development methodology should encourage the existence of formal contracts between developers, so that contracts can be used to reason about correctness of the development process, and to compare the capabilities of various groupings of agents (coalitions) in order to accomplish a particular goal.

We propose to apply the notion of a formal contract described in the previous section, to the software development process itself. That is to say, the software development process can be seen as involving a number of agents (the development team and the software artifacts) carrying out actions with the goal of building a software system that meets the user requirements. The software development process consists of a collection of interacting activities. When specifying a specific activity, we may consider the other activities to be controlled by other agents. We may need some of these activities in order to carry out the set of tasks of our activity, but we cannot influence the choices made by the other agents. This situation is analogous to a contractor using subcontractors.

A specification of an activity is a contract that gives some constraints on the results and effects of the activity but leaves freedom for the agent to decide how the actual behavior is to be realized. For example, a member of the development team, say the agent ai, agrees to take over the task of specifying a method of a given Class by either creating a State machine, or a sequence diagram or a set of pre-and post-conditions,

$$\text{Contract S} = \text{create-SM} \cup_{ai} \text{create-SeqD} \cup_{ai} \text{write-Pre\&Post}$$

The rest of the agents can assume that after ai carries out their contracts, some specification for the method does exists, but they do not know precisely which alternative was chosen; so whatever they want to achieve, it should be achieved no matter which alternative was chosen.

A remarkable difference between traditional software contracts and software process contracts (sp-contracts) is the kind of object constituting a state. While in software contracts, objects in the state represent objects in a software system (e.g., a bank account object in a banking system), in sp-contracts, objects in the state are process artifacts, such as a class diagram or a use case model. But this difference is just conceptual, from the mathematical point of view we can reason about process contracts in the standard way, as if they were software contracts. This view of software process as software is not new, we can go back to the work of Osterweil (1997).

Building sp-Contracts

There are different levels of granularity in which sp-contracts are defined. On the one hand we have contracts regulating primitive evolution, such as adding a single class in a Class diagram, while on the other hand, we have contracts defining complex evolution, such as the realization of a use case in the analysis phase by a collaboration diagram in the design phase, or the reorganization of a complete class hierarchy. Complex evolutions are non-atomic tasks which are composed by a number of primitive tasks. We start specifying atomic contracts (contracts explaining primitive tasks) which will be the building blocks for non-atomic contracts (i.e., regulations for complex evolution activities).

Primitive sp-Contracts

To make contracts more understandable and extensible, we use the object-oriented approach to specify them. The object-oriented approach deals with the complexity of description of software development process better than the traditional approach. Examples of this are the framework for describing UML compatible development processes defined by Hruby (1999) and the metamodel defined by the OMG Process Working Group (OMG, 1998), among others. In the object-oriented approach, software artifacts produced during the development process are considered objects with methods and attributes.

A Class is a template used to describe objects with identical behavior. The Refinement Calculus has been applied to the specification of Classes, by giving a syntax for the Class declaration and a formal semantics for object instantiation, message passing, inheritance and substitutability (Back, Mikhajlova, & von Wright, 1997; Back, Mikhajlov, & von Wright , 2000).

A Class is given by the following declaration:

C = **subclass** of P

var $attr_1$:Σ1,...,$attr_m$:Σm

C(val x_0:Γ_0)= K,

$Meth_1$(val x_1:Γ_1 , **res** y_1:Δ_1) = $\mathbf{M}_1$,

...

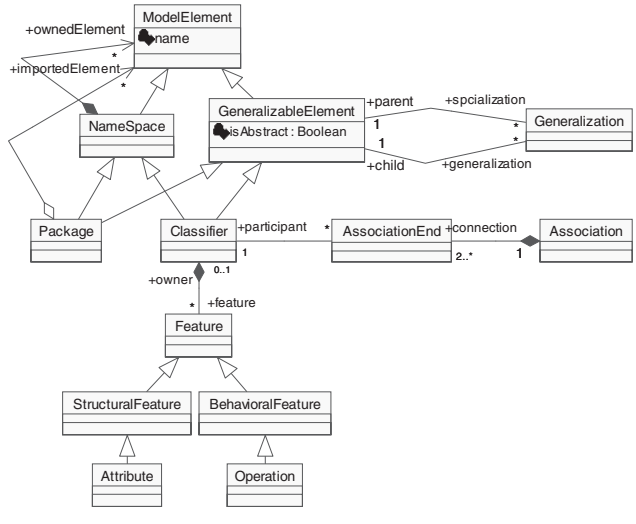
$Meth_n$(val x_1:Γ_1 , **res** y_1:Δ_1) = $\mathbf{M}_n$,

end

This class C describes attributes, specifies the way the objects are created, and gives a (possibly nondeterministic) specification for each method. Class attributes $attr_1$,..,$attr_m$ have the corresponding types Σ_1...Σ_m. The identifier self represents the tuple ($attr_1$,...,$attr_m$). The type of self is $\Sigma = \Sigma_1$ x ... x Σ_m. A class constructor is used to instantiate objects and has the same name as the class. The statement K :$\Gamma_0 \rightarrow \Sigma$ x Γ_0, representing the body of the constructor, introduces the attributes into the state space and initializes them using the input parameter x_0:Γ_0. Methods $Meth_1$... $Meth_n$ specified by bodies M_1 ... M_n operate on the attributes and realize the object functionality. Every statement M_i is of type (Σ x Γ_i x Δ_i) → (Σ x Γ_i x Δ_i). The identifier self acts as an implicit result parameter of the constructor and an implicit variable parameter of the methods.

In general, every body M_i includes a precondition p_i and an effect S_i (M_i=*assert* p_i ; S_i). When a method M_i is called there is an agent ***a*** responsible for the call. The method invocation is then interpreted as the following contract: (assert$_a$ p_i ; S_i), that is, the agent is responsible for verifying the preconditions of the method. If agent ***a*** has invoked the method in a state that does not satisfy the precondition, then ***a*** has breached the contract.

Figure 3. Part of the UML metamodel



For the sake of readability, we write $Meth_i$(**val** $x_i : \Gamma_i$**)** : $\Delta_i = M_i$[resu/y_n], to denote the declaration $Meth_n$(**val** x_n:Γ_1 , **res** y_n:Δ_n) = M_n, . The result variable y_n is replaced by the special variable called *resu* in the body of the method. And regarding method invocation, we write o.$meth_i$(x_i).$meth_j$(x_j) to indicate that the second method is applied on the object that results from the first invocation, that is to say: o.$meth_i$(x_i, z); z.$meth_j$(x_j).

The library of primitive contracts intentionally reflects the class hierarchy of the UML metamodel (OMG, 2003). Contract library consists of a set of UML artifact's specifications (metaclasses), where each specification describes both the artifact's properties (i.e., attributes of the artifact) and all the possible ways of modifying the artifact (i.e., operations that can be applied on the artifact, such as adding a new feature to a class).

Figure 3 shows a part of the UML metamodel. Primitive contracts for these artifacts are (partially) specified as follows:

Generalization = **subclass of** Relationship

var parent, child : GeneralizableElement,

Constructor Generalization(**val** p,c : GeneralizableElement) = parent:=p; child:=c,
parent() : GeneralizableElement = **resu**:=parent,

child() : GeneralizableElement = **resu**:=child,

end

The Class Generalization has an internal state composed by two attributes called parent and child, respectively, both storing a GeneralizableElement. The Class defines a constructor operation and two observer methods, one for each attribute.

NameSpace = **subclass of** ModelElement
var ownedElements : Set of ModelElement,
Constructor NameSpace() = ownedElements:= {} ,
ownedElements() : Set of ModelElement = resu:=ownedElements,
addElement(val e:ModelElement) =
assert (e∉ ownedElements ∧ ∀g• (g∈ ownedElements → e.name ≠ g.name)) ;
ownedElements:= ownedElements ∪ {e} ,
deleteElement(val e:ModelElement) =
assert (e∈ ownedElements) ; ownedElements:= ownedElements - {e} ,
end

GeneralizableElement = **subclass of** ModelElement
var generalizations, specializations : Set of Generalization,
isAbstract: Bool
Constructor GeneralizableElement() = generalizations := {};
specializations := {},

```
parents() :   Set of GeneralizableElement =
                resu:= generalizations.collect(parent)1,
children() : Set of GeneralizableElement =
                resu:= specializations.collect(child),
allParents() : Set of GeneralizableElement =
          ps := self.parents() ; resu := ps ∪ ps.collect(allParents),
isA(val c : GeneralizableElement) : Bool =
                resu:= (self=c ∨ c ∈ self.allParent()) ,
end
```

The Class GeneralizableElement has an internal state composed by three attributes, the first two attributes containing a set of Generalizations and the third attribute containing a Boolean value. The Class defines a set of methods: method parents() returns a Set consisting of all direct parents of the generalizable element which are accesible through its Generalizations; the method children() returns a set of all direct children; the method allParents() results in a Set containing all ancestors. IsA() returns true if the receiver of the message is a subclass (direct or indirect) of the parameter.

```
Feature = subclass of ModelElement
      var owner : Classifier,
      Constructor Feature (val o : Classifier) = owner:=o,
      owner() : Classifier = resu:=owner,
      setOwner(val o:Classifier) = owner:=o,
end

Classifier = subclass of GeneralizableElement, NameSpace
  var features : Set of Feature,
    associationEnds : Set of AssociationEnd,
  Constructor Classifier() = features :={}; associationEnds :={},
  allFeatures() :  Set of Feature =
                    resu:= (features ∪ self.parents.collect(allFeatures) ),
  associations(): Set of Association =
                    resu:= self.association.collect(association),
  oppositeAssociationEnds() : Set of AssociationEnd = ...
      .....
   addFeature(val f : Feature) =
            assert (f ∉ features ∧ ∀g • (g ∈ features ∨
            g ∈ self.oppositeAssociationEnds → f.name ≠ g.name) ) ;
            features:= features ∪ {f} ; f.setOwner(self),
  deleteFeature(val f:Feature) =
                assert f∈self.features;self.features:=self.features - {f}
end
```

The Class Classifier has an internal state with two attributes; the first one stores a set of Features while the second one stores a set of AssociationEnds. The Class defines a

set of query methods: the method allFeatures() results in a Set containing all Features of the Classifier itself and all its inherited Features; the operation associations results in a Set containing all Associations of the Classifier itself; the operation oppositeAssociationEnds results in a set of all AssociationEnds that are opposite to the Classifier. Additionally, the Class declares a set of mutator methods which modify the object internal state: the method addFeature() has a precondition stating that the new feature does not belong to the classifier, and the new feature should have a different name from all the other attributes in the classifier, and from all the opposite associationEnds of the classifier. The effect of the method is that the feature is added to the list of features and the classifier is set as the feature's owner.

```
Package = subclass of NameSpace, GeneralizableElement
        var importedElements : Set of ModelElement,
        Constructor Package() = importedElement:= {} ,
            allContents() : Set of ModelElement = resu:=ownedElement ∪
importedElement,
        addGeneralization(val g:Generalization) =
                assert (g ∉ ownedElements ∧ g.parent ∈ ownedElements ∧
                        g.child ∈  ownedElements ∧ ¬ g.parent.isA(g.child) ;
                self.addElement(g) ,
end
```

The class Package inherits from NameSpace and GeneralizableElement. It specifies a method addGeneralization() to insert a new generalization in the package. The preconditions for the method are that the generalization is not in the package, all elements connected by the new relationship (i.e., the parent and the child) are included in the package and that the new generalization preserves absence of circular inheritance. The effect of the method is that the new element is added in the collection of owned elements of the package by invoking the method addElement() inherited from the class NameSpace.

Apart from software artifact specifications, the other component in the formalism of sp-contract is the specification of software developers. Software developers are specified by declaring their attributes and the contracts for their activities:

```
Developer = subclass of Object
        var name : String , skills : Set of String,
        Constructor Developer(val n : String) = name := n ,
        .....
end
```

The class Developer is the root in the hierarchy, it will be subsequently specialized in order to specify concrete behavior of specific developers.

Complex sp-Contracts

On top of primitive contracts it is possible to define complex contracts, specifying non-atomic forms of evolution through the software development process. Then, by using the **wp** predicate transformer we can verify whether a set of agents (i.e., software developers) can achieve their goal or not. We can analyze whether a developer (or team of developers) can apply a group of modifications on a model or not by means of a contract designed in terms of a set of primitive operations conforming the group.

Developers will successfully carry out the modifications if some preconditions hold. We can determine the weakest preconditions to achieve a goal by computing $\mathbf{wp}_A$. C . Q, where C is the contract, A is the set of software developers (agents) and Q is the goal.

If computing the **wp** we obtain a predicate different from false, then we proved that with the contract the developers can achieve their goal under certain preconditions.

The **wp** formalism allows us to analyze a single contract from the point of view of different coalitions of agents. If computing the **wp** we obtain 'false,' we can look for a different coalition (e.g., we can permit an outside agent to join the coalition) and compare the results. In other case (if the coalition should be preserved) to achieve the goal the contract have to be modified.

In the following sections, we give examples of complex contract.

Example 1: Contract on the Evolution Dimension

Consider a collaborative activity, in which two software developers have to carry out a refactoring on a class diagram. One of the agents will detect and move all the features that could be pulled up to a superclass, while the other agent will simplify the class diagram by collecting empty classes.

To coordinate this collaborative activity, both agents (e.g., the lifter and the cleaner) subscribe a complex contract that is built on top of primitive contracts establishing the primitive responsibilities for each agent.

The primitive specification for the cleaner agent describes a method called deleteEmptyClass(), as follows:

```
Cleaner = subclass of Developer
        deleteEmptyClass(val p : Package) =
                (update_self c:=c´ | c' ∈ p.ownedElement ∧ c'.features=∅ ∧
                                                c'.children=∅∧ c'.associations=∅) ;
                  p.deleteElement(c);
end
```

The contract for this method states that the agent will detect nondeterministically a class c from the package p given as parameter, such that class c is empty (i.e., it has no children and no features). Then the selected class is deleted from the package.

The primitive specification for the lifter agent contains three methods:

Lifter = **subclass of** Developer
pasteRepeatedFeature(**val** c : Class) =
(**update**$_{self}$ f:=f´ | c´∈ c.children • f´∈ c'.features) ; c.addFeature(f)
,
deleteRepeatedFeature(**val** c : Class) =
(**update**$_{self}$ f:=f´ | c´∈ c.children • f´∈ c'.features) ;
for i=1 **to** (c.children.size) **do** c´:=c.children.at(i); c´.deleteFeature(f)
od,
liftRepeatedFeature(**val** p : Package) =
(**update**$_{self}$ c:=s | s ∈ p.ownedElement ∧
∃f:Feature • (c´∈ s.children • f∈ c'.features)) ;
(pasteRepeatedFeature(c); deleteRepeatedFeature(c))
$\cup_{self}$
(deleteRepeatedFeature(c); pasteRepeatedFeature(c)) ;
end

The method pasteRepeatedFeature() says that the agent will receive a class as parameter and will select nondeterministically a feature that appears in all subclasses of the given class. Then, the selected feature is pasted in the class; the method deleteRepeatedFeature() states that the agent will select nondeterministically a feature that appears in all subclasses of a given class. Then, the selected feature is deleted from all the subclasses; finally liftRepeatedFeature() is a more complex method that allows the agent to choose nondeterministically in which order to carry out its activities, after having selected (nondeterministically) a class that is candidate for refactoring).

The complex contract **R** states that both developers (plus a coordinator agent named coord) commit themselves to carry out the refactoring task in a collaborative way. The coordinator agent will nondeterministically choose either asking a_1 to lift a repeated feature or asking a_2 to delete an empty class. The terms of the contract are as follows:

R = a_1 := **new** Lifter ; a_2 := **new** Cleaner ;
while (¬**Q**) **do** a_1.liftRepeatedFeature(p) $\cup_{coord}$
a_2.deleteEmptyClass(p) **od**;

Where **Q** specifies the expected effect of the refactoring activity: (i) there is no repeated feature and (ii) the model does not contain any empty class:

Q = ∀c:Class • c∈ p.ownedElement →
(¬∃f:Feature • (∀c´∈ c.children • f∈ c'.features) ∧ (i)
(c.features≠∅ ∨ c.children≠∅ ∨ c.associations≠∅)) (ii)

We may be interested in calculating the weakest precondition for agents a_1 and a_2 to reach the goal G by using the contract R. That is to say: $\mathbf{wp}_{\{coord,\ a1,\ a2\}}$. R. G

where G = Q∧T , being T a formula specifying that the resulting model keeps all the functionality of the original model.

Applying the calculus is possible to determine that if agents a_1 and a_2 work together (i.e., both of them integrate the coalition), then they can reach the goal.

But if a_1 leaves the coalition, the **wp** is false. The achievement of the goal cannot be guaranteed because agent a_1 is free to resolve their nondeterministic choices in a hostile way. For example, in the following choice:

(pasteRepeatedFeature(c) ; deleteRepeatedFeature(c))
$\cup_{a1}$
(deleteRepeatedFeature(c) ; pasteRepeatedFeature(c))

only the first option guarantees the achievement of the goal. If agent a_1 chooses the second option a problem will occur: A feature is deleted before being pasted in the superclass; consequently, the model loses functionality and the final goal cannot be achieved.

Example 2: Contract on the Horizontal Dimension

Arbitrary modifications that do not cause problems when they are applied exclusively, may originate conflicts when they are integrated. Consider a collaborative task in which two agents a_1 and a_2 need to add a generalization relationship respectively to a model, preserving the consistency of the model.

Contract statement **C** specifies that agents a_1 and a_2 will perform their activities sequentially, one after the other:

C = a_1 := **new** Designer ; a_2 := **new** Designer ;
a_1.addGeneralization(p,r) ; a_2.addGeneralization(p,g)

The primitive contract regulating the behavior for Designer states that any designer will accomplish this task by directly invoking the method addGeneralization() of the package artifact:

Designer = **subclass of** Developer
addGeneralization(**val** p : Package , g: Generalization) = p.addGeneralization(g)
end

As we explained before, the method invocation is interpreted as the following contract: ($assert_{al}$ p_i ; S_i), that is, the agent takes over the responsibility for the preconditions of

the method. If agent a_1 (respectively a_2) invokes the method in a state that does not satisfy the precondition, then a_1 breaches the contract.

Using the calculus, it is possible to find out which is the weakest precondition to achieve the goal of introducing two generalization relationships without breaking the non-circularity principle of inheritance hierarchies by computing: $\mathbf{wp}_{\{a1,a2\}}$. C . Q, where C is the contract between agents and Q is the post-condition that specifies the goal of the activity, which is the creation of new generalization relationships guaranteeing the absence of circularity in the class hierarchy:

$$Q = (r \in p.ownedElements \wedge g \in p.ownedElements) \wedge$$
$$\forall c_1,c_2 : GeneralizableElement.\ (c_1.isA(c_2) \wedge c_2.isA(c_1) \rightarrow c_2 = c_1)$$

The weakest precondition P for agents a_1 and a_2 to reach the goal Q by using the contract C, (i.e., P = $\mathbf{wp}_{\{a1,a2\}}$. C. Q) can be semi automatically calculated applying the rules in section 2.3 arriving to the following result:

P =

(i) $r \notin p.ownedElements \wedge r.parent \in p.ownedElements \wedge$
$r.child \in p.ownedElements \wedge \neg\ r.parent.isA(r.child) \wedge$

(ii) $g \notin p.ownedElements \wedge g.parent \in p.ownedElements \wedge$
$g.child \in p.ownedElements \wedge \neg\ g.parent.isA(g.child) \wedge$

(iii) $\forall c_1,c_2 : GeneralizableElement.(c_1.isA(c_2) \wedge c_2.isA(c_1) \rightarrow c_2 = c_1)$

(iv) $\neg\ (g.parent.isA(r.child) \wedge r.parent.isA(g.child))$

Where (i), (ii) and (iii) specify the precondition for applying the first and the second evolution, respectively (as if they were applied in isolation), and (iv) specifies a special requirement to avoid circular inheritance in the case that both evolution actions were applied together.

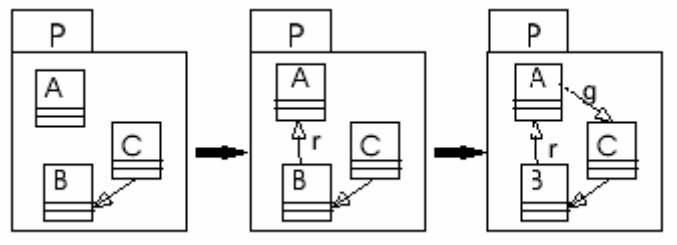
Figure 4 illustrates a conflictive case, in which the expected weakest precondition does not hold in the initial state. As a consequence agents cannot achieve their goals (because a circularity is introduced) in spite of fulfilling the contract.

Future Trends

The sp-contract formalism should be equipped with automatic tools supporting contract derivation, precondition calculation, and correctness calculation. These tools should be connected with the Refinement Calculator (Butler et al., 1997; Celiku & von Right, 2002), which supports the Refinement Calculus (Back & von Wright, 1998).

The need for automatic support is the main motivation for future work. It is necessary to count with a tool to assist developers in the task of writing and applying sp-contracts.

Figure 4. Collaborative evolution breaching consistency rule



This tool should be integrated with an environment for thesoftware development process, providing the following functionality:

- **Contract edition and storage:** Developers can create new contracts and store them in a repository. There are different ways in which a new contract can be created: from scratch as a primitive contract; by specializing an existing contract stored in the repository; or by selecting a group of contracts stored in the repository and composing them to form a new complex contract.
- **Pre condition calculus:** Given a contract between a set of agents and a specific goal, the tool would be able to compute the weakest precondition for the application of that contract.
- **Contract refinement:** Specific contracts can be derived from abstract contracts by applying the refinement calculus.
- **Contract correctness:** If a precondition p and a post-condition q are given and contract S has already been defined, we can prove that S is correct with respect to precondition p and post-condition q.
- **Visual assistance:** functions of edition of contracts have a textual interface using mathematical notation, but also may have a graphical interface. A contract can be created using a UML editor that both records the operations applied on models (such as adding a new class) and translates them to the mathematical notation. This translation is straightforward using the primitive contracts on the UML artifacts described. On the other hand, the task of selecting a contract from the repository will be assisted by the generation of a graphical view of the contract. This is provided by animating a contract, that is to say, showing how the execution of the contract modifies a given UML model, step by step.

Conclusion and Related Work

During the software development process different UML models are employed to specify the system from different viewpoints at different levels of abstraction. Models of

different viewpoints have a certain overlap (Spanoudakis, Frinkelstein, & Till, 1999) and models produced at different levels of abstractions in the development process also are related. Consequently, handling of consistency between models is of major importance (Ghezzi & Nuseibeh, 1999; Kuzniarz, Huzar, Reggio, & Sourrouille, 2002; Kuzniarz, Huzar, Reggio, Sourrouille, & Ataron, 2003).

Different types of consistency problems have been identified; Engels, Küster, Heckel, and Groenewegen (2001) distinguish two dimensions of consistency problems — horizontal and vertical:

- Horizontal consistency concerns specifications consisting of different parts representing the different points of view from which the system is specified.
- Vertical consistency arises when a model is transformed into another refined model. For example a collaboration diagram can be derived from a use case diagram.

However, we need to distinguish three dimensions of consistency (Pons, Giandini, & Baum, 2000): Horizontal, Vertical, and Evolution dimensions (the last two refine Engels' vertical dimension) because two dimensions are insufficient to comprise an iterative and incremental software process where model refinement occurs vertically, inside each iteration; but also horizontally, from one iteration to the next one.

A wide range of different approaches for checking consistency of UML models has been proposed in the literature. Here is an overview of the most relevant works, classified in two groups. The first group focuses on the consistency between a fixed set of artifacts:

Glinz (2000) defines a lightweight approach to consistency between a scenario model and a class model. He assumes semi-formal, loosely coupled models that are complementary: scenarios model the external system behavior, the class model specifies the internal functionality. He achieves consistency by minimizing overlap between the two models and by systematically cross referencing corresponding information. He gives a set of rules (some of them automatically checked) that can be used both for developing a consistent specification and for checking the consistency of a completed specification.

Petriu Sun (2000) analyze the consistency between two different UML sublanguages: Activity diagrams and Sequence Diagrams.

Whittle and Schumann (2000) developed an algorithm for automatically generating statechart designs from a collection of sequence diagrams.

Ehrig and Tsiolakis (2000) investigate the consistency between UML class and sequence diagrams by representing them by attributed graph grammars.

Works in the second group propose a general methodology that can be applied to different consistency problems:

Astesiano and Reggio (2003) look at the consistency problems in the UML in terms of the well-known machinery of classical algebraic specifications. Thus, first they review how the various kinds of consistency problems were formulated in that setting. A similar approach, but using dynamic logic, was defined by Pons and Baum (2000).

Engels et al. (2001) discuss the issue of consistency of behavioral models in the UML and present a general methodology about how consistency problems can be dealt with. According to the methodology, those aspects of the models relevant to the consistency are mapped to a semantic domain in which precise consistency tests can be formulated. The choice of the semantics domain and the definition of consistency conditions vary according to each concrete consistency problem. An instantiation of this approach is the work of Fradet, Le Métayer, and Périn (1999) where systems of linear inequalities are used to check consistency for multiple view software architectures. The general idea is further enhanced in Engels, Heckel, Küster, and Groenewegen (2002) with dynamic metamodeling rules. Model transformation rules are used to represent evolution steps, and their effect on the overall model consistency is explored.

Egyed (2001) presents an approach for automated consistency checking among UML diagrams, called ViewIntegra. The approach makes use of consistent transformation to translate diagrams into interpretations to bring models closer to one another in order to simply comparison.

Grundy, Hosking, and Mugridge (1998) claim that a key requirement for supporting inconsistency management is the facilities for developers to configure when and how inconsistencies are detected, monitored, stored, presented and possibly automatically resolved. They describe their experience with building complex multiple-view software development tools supporting inconsistency management facilities.

Toval and Alemán (2000) formalize the UML notation and transformations between different UML models within rewriting logic. They implement their formalization in the Maude system, focussing on using reflection to represent and support the evolution of models.

Van Der Straeten, Mens, Simmonds, and Jonckers (2003) propose and validate an approach to detect and resolve inconsistencies between different versions of a UML model, specified as a collection of class diagrams, sequence diagrams, and state diagrams. The formalism used is description logic, a decidable fragment of first-order predicate logic. Logic rules are used to detect and to suggest ways to resolve inconsistencies.

The proposal described in this chapter belongs to the second group; sp-contract is a mathematical tool the objective of which is to improve the formality of software development processes. The core of sp-contracts is the formalization of UML software artifacts and their relationships on three dimensions. Sp-contracts handle consistency between models through evolution by specifying state invariant and pre- and post-conditions for each software development task. This feature is closely related to the mechanism of reuse contracts (Steyaert, Lucas, Mens, & D'Hondt, 1996; Mens, Lucas, & D'Hondt, 2000). A reuse contract describes a set of interacting participants. Reuse contracts can only be adapted by means of reuse operators that record both the protocol between developers and users of a reusable component and the relationship between different versions of one component that has evolved.

The originality of sp-contracts resides in the fact that software developers are incorporated into the formalism as agents (or a coalition of agents) who make decisions and have responsibilities (Pons & Baum, 2001; Pons & Baum, 2002). Given a specific goal that a

coalition of agents is requested to achieve, we can use traditional correctness reasoning to show that the goal can in fact be achieved by the coalition, regardless of how the remaining agents act. The weakest precondition formalism allows us to analyze a single contract from the point of view of different coalitions and compare the results. For example, it is possible to study whether a given coalition A would gain anything by permitting an outside agent b to join A. On the other hand, formal refinement techniques can be applied to a contract in order to obtain an improved contract preserving its correctness.

We believe the formalism of sp-contracts can play an important role in the study of software development process: sp-contracts can be useful for reasoning about and justifying good practices in software process, providing a formal rational for them; sp-contracts can provide a means to analyze and reason about refactoring tasks, refinements, and transformation of models.

Regarding scalability issues, when the software development process becomes complex, the formalism allows us to manage the complexity by means of a hierarchical definition and classification of contracts. On the one hand, the library of contracts is organized into a generalization-specialization hierarchy. Then, it is possible to define a new contract by specializing an existing one, by writing only the incremental features. On the other hand, contracts can be specified in a compositional way. It means that complex contracts are built in terms of less complex ones, and weakest preconditions for a complex contract are calculated from weakest preconditions of its constituent contracts. Furthermore, specifications are organized along three different dimensions (horizontal, vertical, and evolution dimension), thus increasing the cohesion and readability of each contract.

References

Andrade, L.F. & Fiadeiro, J.L. (1999). Interconnecting objects via contracts. *Proceedings of the Second International Conference on the Unified Modeling Language. Lecture Notes in Computer Science 1723.* Springer.

Astesiano, E. &, Reggio, G. (2003). *An algebraic proposal for handling UML consistency*. Workshop on Consistency Problems in UML-based Software Development. Blekinge Institute of Technology Research Report 2003:06.

Back, R. & von Wright, J.(1998). *Refinement calculus: A systematic introduction, graduate texts in computer science,* Springer Verlag.

Back, R., Mikhajlova, A. & von Wright, J. (1997). *Class refinement as semantics of correct subclassing.* Turku Centre for Computer Science. TUCS Technical Report No 147. ISBN 952-12-0114-2. December 1997.

Back, R., Petre, L. & Porres Paltor, I. (1999). Analysing UML use cases as contract. *Proceedings of the Second International Conference on the Unified Modeling Language. Lecture Notes in Computer Science 1723.* Springer Verlag.

Back, R., Mikhajlov, L. & von Wright, J. (2000). *Formal semantics of inheritance and object substitutability.* Turku Centre for Computer Science. TUCS Technical Report No 337. ISBN 952-12-0637-3. January 2000.

Butler, M., Grundy, J., Langbacka, T., Ruksenas, R. & Von Wright, J. (1997). The refinement calculator – Proof support for program refinement. *Proceeding of Formal Methods Pacific*. Springer-Verlag.

Celiku, O. & von Right, J. (2002). Theorem prover support for precondition and correctness calculation. *Proceedings of the Fourth 4th International Conference on Formal Engineering Methods ICFEM (LNCS 2495).* Springer-Verlag.

Egyed, A. (2001, November). Scalable consistency checking between diagrams – The VIEWINTEGRA approach. *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE),* San Diego.

Ehrig, H. & Tsiolakis, A. (2000). Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In H. Ehrig & G. Taentzer (Eds.), *ETAPS 2000 Workshop on Graph Transformation Systems* (pp. 77-86).

Engels,G., Heckel, R., Küster, J. & Groenewegen, L. (2002). Consistency-preserving model evolution through transformations. In *Proceedings of the International Conference on. The Unified Modeling Language. Model Engineering, Concepts, and Tools, number 2460 in Lecture Notes in Computer Science* (pp. 212-227). Springer-Verlag.

Engels, G., Küster, J., Heckel, R., & Groenewegen, L. (2001). A methodology for specifying and analyzing consistency of object oriented behavioral models. *Proceedings of the IEEE International Conference on Foundation of Software Engineering,* Vienna.

Fradet, P., Le Métayer, D., & Périn, M. (1999). Consistency checking for multiple view software architectures. In *Proceedings of the International Conference on ESEC/FSE'99, volume 1687 of Lecture Notes in Computer Science* (pp. 410-428). Springer-Verlag.

Ghezzi, C. & Nuseibeh, B. (1999). Special Issue on Managing Inconsistency in Software Development (2). *IEEE Transaction on Software Engineering, 25*(11).

Glinz, Martin. (2000). A lightweight approach to consistency of scenarios and class models. *Proceedings of the Fourth International Conference on Requirements Engineering,* Schaumburg, IL, June 10-23.

Grundy, J.C, Hosking, J.G., & Mugridge, W.B. (1998). Inconsistency management for multiple-view software development environments. *IEEE Transactions on Software Engineering, 24*(11), 960-981.

Helm, R. Holland, I., & Gangopadhyay, D. (1990). Contracts: Specifying behavioral compositions in object-oriented systems. *Proceedings of OOPSLA '90.* ACM Press.

Hruby, Pl. (1999). Framework for describing UML compatible development processes. *Proceedings of the Unified Modeling Language Conference. Lecture Notes in Computer Science 1723.* Springer.

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The unified software development process* (Professional, 1st edition). Addison Wesley.

Kuzniarz, L., Huzar, Z., Reggio, G., & Sourrouille, J. (Eds.) (2002). *Proceedings of the first Workshop on "Consistency Problems in UML-Software Development."* RR-2002-6.

Kuzniarz, L., Huzar, Z., Reggio, G., Sourrouille, J., & Ataron, M. (Eds.). (2003). *Proceedings of the Second Workshop on "Consistency Problems in UML-Software Development."* Blekinge Institute of Technology Research Report 2003:06.

Mens, T., Lucas, C., & D'Hondt, T. (2000). Automating support for software evolution in UML. *Automated Software Engineering Journal, 7,* 1.

Meyer, B. (1992). *Advances in object oriented software engineering.* (Chapter 1, Design by Contract). Prentice Hall.

Meyer, B.(1997). *Object-oriented software construction* (2nd edition). Prentice Hall.

OMG (1998, July). *Analysis and design process engineering.* Process Working Group, Analysis and Design Platform Task Force.

OMG (2003, March). The unified modeling language specification version1.5, revised by the Object Management Group. Available online at *http://www.omg.org*

Osterweil, L. (1997). Software processes are software too. Revisited: An invited talk on the most influential paper of ICSE. *Proceedings of the 19th International Conference on Software Engineering.* ACM Press.

Overgaard, G. & Palmkvist, K.(2000). Interacting subsystems in UML. In *Proceedings of The Third International Conference on the Unified Modeling Language. Lecture Notes in Computer Science.* Spring Verlag.

Petriu, D. & Sun, Y. (2000) Consistent behaviour representation in activity and sequence diagrams. In *Proceedings of the Third International Conference on the Unified Modeling Language. Lecture Notes in Computer Science.* Spring Verlag.

Pons, C. & Baum, G. (2000). Formal foundations of object-oriented modeling notations. *The Third International Conference on Formal Engineering Methods, ICFEM 2000,* York, UK. IEEE Computer Society Press.

Pons, C. & Baum, G. (2001). Software development contracts. *The 5th European Conference on Software Maintenance and Reengineering, Special Session on Formal Foundation of Software Evolution.* Portugal.

Pons, C. & Baum, G. (2002). Contracts soundness for object oriented software development process. *OOPSLA2002 Workshop on Behavioral Semantics.* Seattle, WA. Northeastern University, College of Computer Science, 163-177.

Pons, C., Giandini, R., & Baum, G. (2000). Dependency relationships between models through the software development process. *The 10th International Workshop on Software Specification and Design (IWSSD),* California. IEEE Computer Society Press.

Spanoudakis, G., Frinkelstein, A., & Till, D. (1999). Overlaps in requirement engineering. *Automated Software Engineering: An International Journal, 6*(2), 171-198.

Steyaert, P., Lucas, C., Mens, K., & D'Hondt, T. (1996). Reuse contracts: Managing the evolution of reusable assets. *Proceedings of OOPSLA'96,* New York. ACM Press.

Toval, A. & Alemán, J. (2000). Formally modeling UML and its evolution: A holistic approach. In S. Smith & C. Talcott (Eds.), *Formal methods for open object-based distributed systems IV* (pp. 183-206). Kluwer Academic Publishers.

Van Der Straeten, R., Mens, T., Simmonds, J., & Jonckers, V.(2003) Using description logic to maintain consistency between UML-models. *Proceedings of the Sixth International Conference on the Unified Modeling Language. Lecture Notes in Computer Science number 2863.* Springer.

Van Gorp, P., Stenten, H., Mens, T., & Demeyer, S. (2003). Towards automating source-consistent UML refactoring. *Proceedings of the Sixth International Conference on the Unified Modeling Language. Lecture Notes in Computer Science number 2863.* Springer.

Whittle, J. & Schumann, J. (2000). Generating statechart designs from scenarios. *Proceedings of International Conference on Software Engineering ICSE 2000.* Limerick, Ireland.

Endnote

[1] The type Set provides the traditional operations: select, reject, collect (or map), size, $\cup$, $\cap$, $-$, $\in$, $\subseteq$.

Chapter X

Visualising COBOL Legacy Systems with UML: An Experimental Report

Steve McRobb, De Montfort University, UK

Richard Millham, De Montfort University, UK

Jianjun Pu, De Montfort University, UK

Hongji Yang, De Montfort University, UK

Abstract

This chapter presents a report of an experimental approach that uses WSL as an intermediate language for the visualisation of COBOL legacy systems in UML. Key UML techniques are identified that can be used for visualisation. Many cases were studied, and one is presented in detail. The report concludes by demonstrating how this approach can be used to build a software tool that automates the visualisation task. Furthermore, understanding a system is of critical importance to a developer who must be able to understand the business processes being modeled by the system along with the system's functionality, structure, events, and interactions with external entities. Such an understanding is of even more importance in reverse engineering. Although developers have the advantage of having the source code available, system documentation is often missing or incomplete, and the original users, whose requirements were used to design the system, are often long gone.

Introduction

A developer requires a model of a system not only in order to understand the business processes being modeled but also the structure and dynamics of the system. Visualisation of the system model is often necessary in order to clearly depict the complex relationships among model elements of that system.

This chapter focuses on the visualisation of a system from a reengineering, rather than a forward engineering, point of view as an attempt is made to extract documentation from the system code into the visual notation of UML. In many legacy systems, the system code is the only surviving artefact and consequently, any reengineering efforts must focus on this artefact. Visualisation, whether through UML or some other graphical notation, is important because it better enables our brains to make the connections between the software system and the ideas represented within this system; this connection is much more difficult if the documentation is based purely on textual form. This chapter also investigates whether it is possible to visualise a system and which of the nine possible UML diagrams are needed to represent this visualisation. A detailed outline of the process of deriving information from an analysis of the selected legacy system, a batch-oriented system, is given, along with the rules to convert this information to a UML model of this system. A brief overview of the methods, along with their inherent difficulty that are used to extract UML diagrams from a legacy system is provided. Given the selected system, a batch-oriented legacy system, and relying on the only surviving form of documentation, it was found that it was not possible to extract statecharts or use cases from this system. Use cases, which model external actors and business processes of a system, cannot satisfactorily be extracted from system code alone but require intensive user intervention and guidance in order to identify the external actors and their roles. Statecharts, which model external events and the system's response to them, cannot be extracted satisfactorily from a batch-oriented legacy system for two reasons. One reason is that batch-oriented systems have very few external events; usually, their only external event is the arrival of input. The other reason is that an event-response trace of the system, which can be used to model statecharts of this system, cannot be obtained through analysis of system code alone; this event-response trace requires the use of a run-time environment along with a full set of possible external events that this system might encounter. Finally, a tool, TAGDUR, is introduced which has automated some of these analysis processes and which models some aspects of this system in UML.

Why Visualisation of the Legacy System is Necessary for Reverse Engineering

Program understanding can be defined as the process of developing an accurate mental model of a software system's intended architecture, purpose, and behaviour. This model is developed through the use of pattern matching, visualisation, and knowledge-based techniques. The field of program understanding involves fundamental issues of code representation, structural system representation, data and control flow, quality and

complexity metrics, localisation of algorithms and plans, identification of abstract data types and generic operations, and multiple view system analysis using visualisation and analysis tools. Reverse engineering involves analysing data bindings, common reference analysis, similarity analysis, and subsystem and redundancy analysis (Whitney, 1995).

Program understanding involves the use of tools that perform complex reasoning about how behaviours and properties arise from particular combinations of language primitives within the program. One method of program understanding is to use visitors, or small reusable classes, whose function is to parse the source system and evaluate the combinations of language primitives that had been discovered during parsing (Wills, 1993). Other methods try to evaluate and understand a system by taking, as input, the goals and purpose of the system as a specification (Johnson, 1986). Another method is to use clichés that try to recognise commonly used data structures and algorithms and then match these structures and algorithms to higher level abstractions. Examples of clichés are the structures and algorithms associated with hash tables and priority queues. The degree of accuracy of the matching varies with the goal of program understanding. For example, an exact match is needed for program verification while only a reasonably close match is needed for documentation purposes.

Software visualisation is a technique to enable humans to use their brain to make analogies and to link a visual software representation with the ideas that this representation portrays. This link would be much more difficult to make if the software representations were purely in textual form. Software visualisation relies on crafts such as animation, graphic design, and cinematography (Price, Small I, & Baecker, 1992).

Software visualisation has been used for decades in order to help developers understand programs. These techniques vary from flowcharts (Goldstein & Neumann, 1949) to animated graphical representations of data structures (Baecker, 1981). However, many of these software visualisation systems are limited to displaying one type of data or level of abstraction. Few visualisation systems have the ability to suppress lower level detail in order to depict higher level concepts of the system.

Program visualisation systems or tools can be characterised according to their scope, content, form, method, interaction, and effectiveness (Price et al., 1992). Scope refers to the visualisation system's general characteristics such as whether it models concurrent programs or whether there are any size limitations as to the system being depicted. Content refers to the content being visualised. Some visualisation systems can model both data and code, while others model algorithms only. Form refers to what elements are being used in the visualisation. Some visualisation systems use animated graphics, while other systems provide multiple views of different parts of the system. Method refers to how the tool specifies the visualisation. Does the tool require the program source code to be modified in order for it to be visualised? Some tools require the user to insert special statements in code of special interest in order for this code to be properly visualised. Interaction refers to how the user interacts and controls the visualisation. How does the user navigate through the visualisation of a large system in order to see how different parts of the system are being modeled? Effectiveness refers to how well the visualisation communicates information regarding the system being visualised (Price et al., 1992).

Using this taxonomy, a number of program visualisation systems can be classified and grouped in order to enable the developers to find the visualisation tool that best fits their needs. For example, if the source programming language is Pascal, Balsa might be a better visualisation tool for the developers than LogoMotion. Some examples of visualisation tools include the film, Sorting Out Sorting, which is an animated visualisation tool to explain algorithms. Balsa, another visualisation tool, generates animations of Pascal programs. LogoMotion allows users to indicate what aspects of a Logo program they wish to have visualised (Price et al., 1992).

Chifosky and Cross (1990) define reverse engineering to be "the process of analysing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction." Chifosky and Cross state six goals of reverse engineering:

- controlling complexity
- generating alternative views
- recovering lost information
- detecting side effects
- synthesising higher abstractions
- facilitating reuse (Chifosky & Cross, 1990)

A reverse engineering tool, TAGDUR, which is outline later in this chapter, tries to accomplish at least three of these goals. TAGDUR tries to control complexity by encapsulating formerly globally scoped variables and procedures into classes. TAGDUR tries to synthesise higher abstractions by trying to incorporate abstractions, classes, into various abstract representations, such as sequence diagrams, to model system behaviour. TAGDUR generates alternative views by producing UML diagrams, such as sequence or activity diagrams, to model the structure and behaviour of the system.

Creating a model of a system is a useful way to understand the system for many reasons. One reason is that the system itself is a model of an external business process and as such this model can be analysed to ensure that the model accurately maps these external processes. Another reason is that programs are constructed from executable and data structures — these existing entities can be extracted and analysed, through reverse engineering, in order to produce a model of the structure and dynamics of the system (Hall, 1992; Rugaber & Clayton, 1993). Another reason is that a model obtained through reverse engineering, because it embodies explicit knowledge representation of a software system (Van, 1992; Rajlich, 1992), is better able to predict reverse engineering's expected results.

This knowledge representation may be in multiple formats such as textual or graphical notation. Graphical notation has the advantage over textual formats that graphical notations can more clearly depict complex relationships between model elements, such as class (Rugaber & Clayton, 1993).

Why Not Rely on Source Code for System Documentation?

Relying on source code solely to obtain an understanding of the system has many disadvantages, particularly for legacy systems. In many legacy systems, the original design of the system has been obfuscated by the many incremental changes during the system's long maintenance history. Furthermore, the end-users, whose requirements originally helped design the system to meet their business needs, are usually long gone and the documentation outlining these requirements is often missing. Without these original end-users and no documentation, it is often difficult to determine the exact business processes that these systems model.

Source code is very programming-language-dependent (Yang, 1999). In order to understand the code, the developer must be fully proficient in the programming language used to develop the system. The function and role of each section of source code within the system may be obvious to a developer but may be meaningless to a nontechnical end-user. End-users want to see how the business processes that the system represents are modeled and they want to ensure that all of their business requirements are met in the system. End-users are not concerned with the internal design and details of this system.

It is difficult for developers to view parallel data and control flows from reading the code. The control logic, especially if this control logic is heavily nested, is difficult to visualise from the source code, particularly to quickly identify which control constructs affect which parts of code. It is difficult to visualise events occurring in various parts of code and to visualise how these events interact with various objects in the system. Relying on source code as the only documentation source makes it difficult to view the interaction of system objects with other objects and external actors.

Source code encompasses many perspectives such as objects, deployment of components, and timing of object interactions within itself. These multiple perspectives are confusing — it is difficult to represent the source code in each separate perspective. Source code has the additional disadvantage in that it is difficult to represent abstract concepts and behaviour from low-level, detailed source code.

UML

One method to overcome this problem of requiring multiple perspectives of the same system is to visualise the system using some sort of graphical notation. Each perspective is given its own diagram type that is specialised to best represent this perspective.

Rumbaugh, Blaha, Premerlani, Eddy, and Lorenson identify three viewpoints necessary for understanding a software system: the objects manipulated by the computation (the data model), the manipulations themselves (functional model), and how the manipulations are organised and synchronised (the dynamic model).

Rugaber and Clayton state that most of the representation techniques emphasise one of these views.

Representation	References
Object-Oriented Frameworks	(Johnson & Foote, 1988)
Category Theory	(Srinivas, 1991)
Concept Hierarchies	(Biggerstaff, 1989; Lubara, 1991)
Mini-languages	(Neighbors, 1984; Arango, Baxter, Freeman, & Pidgeon, 1986)
Database Languages	(Chen & Ramanoorthy, 1986)
Narrow Spectrum Languages	(Webster, 1987)
Wide Spectrum Languages	(Wile, 1987; Ward, Calliss, & Munro, 1989)
Knowledge Representation Text	(Barstow, 1985)

UML, through its various UML diagrams, encompass all of Rumbaugh's views. The data model is represented by UML's class and object diagrams. The functional model is represented by activity and state diagrams. The dynamic model is represented by sequence and collaboration diagrams.

One of the most common graphical notations is Unified Modeling Language (UML). UML provides multiple perspectives of the system. Use case diagrams model the business processes embodied in the system from a user's perspective. Statecharts and class diagrams model the behaviour and structure of the system, respectively; the behaviour and structure of the system would be of most interest to developers.

UML has many advantages for a graphical modeling notation. UML encompasses many earlier modeling notations that are well understood and well accepted by the software development community. UML has a core set of concepts that remain unchanged, but UML provides a mechanism to extend UML to new concepts and notations beyond this core set. UML allows concepts, notations, and constraints to be specialised for a particular domain.

UML is implementation and development process independent (D'Souza & Wills, 1999). In other words, UML is not dependent on any particular programming language nor is it dependent on a particular development process, such as the waterfall software development model.

UML addresses recurring architectural complexity problems using component technology, visual programming, patterns, and frameworks. These problems include the physical distribution, concurrency and concurrent systems, replication, security, load balancing, and fault tolerance.

In order to enable the developers to achieve the same understanding and interpretation of the model when this model is exchanged among different technologies and tools, some sort of common model exchange format is needed. In order to achieve syntactic interoperability of the model, a canonical mapping of the abstract data representation model to the data interchange format is needed. UML provides several methods to exchange models between tools including UXF (Suzuki, 1999) and XMI (Irwin &

Churcher, 2002). Because UML is general purpose, expressive, simple, widely accepted, and extensible, UML has wide applicability and usability (Muller, 1997).

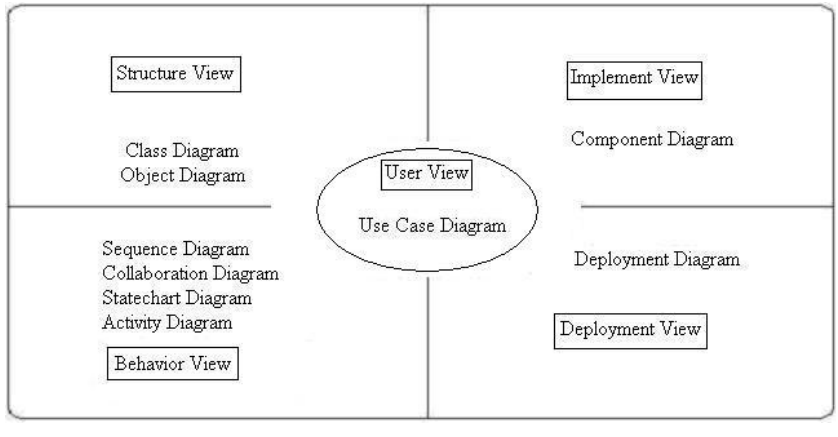
UML Diagrams

UML has different types of diagrams:

- use case
- class
- object
- sequence
- collaboration
- statecharts
- activity
- component
- deployment

Figure 1 embodies the different views of the system in a slightly different way than Rumbaugh's three views of the system. Rumbaugh's static view is embodied in its structured view. Rumbaugh's dynamic and behavioural view is incorporated into a single view, the behavioural view, which describes both the behaviour of the system and the interaction of objects within that system (interactive view).

Figure 1. Different UML diagrams model different views (Alhir, 1998)



Our architectural view is divided up, in Figure 1, into an external system view (the Deployment View), which embodies the deployment diagram, and an internal system view (the Implementation View), which embodies the component diagram. Neither Rumbaugh nor our additional view, the architectural view, includes the user view, which is embodied in use case diagrams, because the user view is usually considered to be external to the system. Figure 1 demonstrates that different authors may classify views of the system differently and accord the various UML diagrams to their particular views differently.

Use case diagrams emphasise services and operations that a system offers to entities outside the system. Use case diagrams are often used to model the business processes that the system represents.

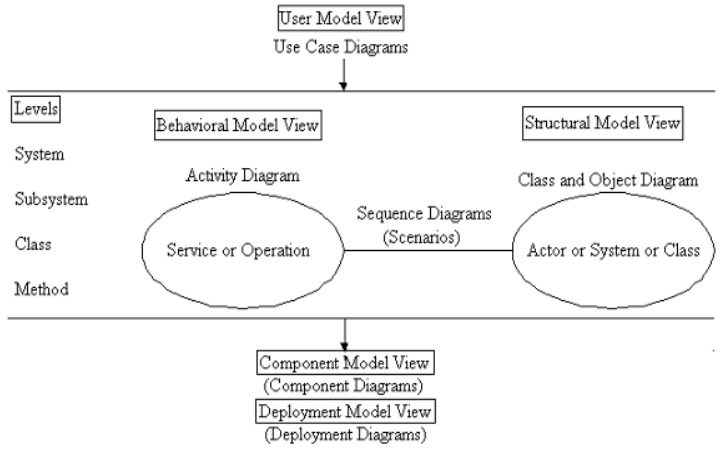
Class diagrams emphasise the static structure of the system. Object diagrams model the static structure of a system at a particular point in its execution. Object and class diagrams can be differentiated in that classes model the structure of a system, while objects are specific examples of the structure.

Sequence diagrams model the messages exchanged over time among the objects within a system. Collaboration diagrams model messages exchanged over time among the objects and their links within a system.

Statecharts model how an object changes or responds to external stimuli within a system. Statecharts model the changes of state embodied within a system due to messages. Activity diagrams model how an object changes or responds to internal processing within a society. Activity diagrams model the flow of control and of information within a system.

Component diagrams model the packaging of an object as a solution. Deployment diagrams model the deployment of objects within a society as a solution within an environment (Muller, 1997).

Figure 2. Different UML diagrams model different views at different levels of abstraction (Muller, 1997)



Which UML Diagrams are Needed to Fully Represent a System?

The nine diagrams of UML represent several different perspectives and several different views. UML diagrams are designed to portray information deemed to be of most interest to a particular set of users. Use case diagrams, which depict the external business environment and business processes modeled by the system, are of most interest to end-users who are not interested in the system's structure, behaviour, or deployment but are interested in ensuring that their business processes are fully represented in the modeled system. Deployment and component diagrams are of most interest to system architects who typically are concerned with the deployment of components within a specific architectural framework rather than the lower level details of structure and behaviour. Developers, on the other hand, are most interested in the internal structure and behaviour of the system.

The question of how many of these nine UML diagrams are needed to properly represent a system is dependent on several factors, including the type and size of the system. Large systems, which are composed of multiple software components, require component and deployment diagrams to reduce this complexity and to depict information most useful to system architects. In smaller systems, with only a few software systems, these components can more easily be kept track of, and a separate diagram, in the form of a deployment diagram, is not needed to depict their particular deployment.

Class diagrams are necessary to describe the structure of a system. Object diagrams may not be necessary for systems with statically created objects; however, object diagrams may be necessary to reduce a system's complexity in a system with dynamically created objects. Object diagrams, in this case, may be needed to depict which objects are active at a particular point in time.

Activity diagrams and statecharts have different focuses on modeled system. Statecharts differ fundamentally from activity diagrams in that statecharts model external events interacting with system objects, but activity diagrams model the internal processing of the system. While external events may be a critical part of the legacy systems that are highly interactive and reactive, many legacy systems are batch-oriented. Because these systems are batch-oriented, the only event external to this type of system is the arrival of batch input that invokes the batch application. Consequently, in batch-oriented legacy systems, activity diagrams are sufficient to describe the behaviour of this system, and these types of systems do not need statecharts to describe their behaviour. However, activity diagrams, because they do not model external events as well as statecharts, are inadequate to describe the behaviour of reactive systems; statecharts must be used as well to model these reactive systems.

Sequence and collaboration diagrams are used to help describe the dynamic behaviour of a system. These diagrams are most useful to help reduce complexity in highly interactive systems with complex behaviour and interactions among objects in the system. However, in non-interactive, strictly batch-oriented systems, the interactions among objects are much less complex and often can be adequately modeled using activity and class diagrams without the need for sequence/collaboration diagrams.

Use case diagrams are very necessary in order to clarify system requirements from the end-users. However, many other methods, such as textual representation or formal specifications methods such as Z, exist to depict the information contained within a use case diagram. Thus, while defining system requirements is a necessary part of the software development process, it is not strictly necessary to utilise use case diagrams to model this information.

Use case diagrams are very necessary in order to clarify system requirements from the end-users. However, many methods exist to depict the information contained within a use case diagram. Thus, while defining system requirements is a necessary part of the software development process, it is not strictly necessary to utilise use case diagrams to model this information.

Why Use UML Diagrams to Represent a Legacy System?

It is necessary to comprehend the legacy system with UML activity diagrams in order to fully comprehend them and enable the developers to make needed changes to them. Software systems are becoming larger and more complicated with the rapidly developing changes in the business world and incremental maintenance over the system's history. Meanwhile, as time marches on and technology changes, these systems tend to be old-fashioned and become legacy. The legacy system may be characterised as a large, complicated, old, heavily modified, difficult to maintain, and old-fashioned software that is still vital to the organisation (Yang, 1999). A legacy system is a computer system or application program that continues to be used because of the cost of replacing or redesigning it, despite its poor competitiveness and compatibility with modern equivalents (Howe, 2002). The implication of the legacy system is that the system is large, monolithic, and difficult to modify. If legacy software only runs on antiquated hardware, the cost of maintaining legacy software may eventually outweigh the cost of replacing both the software and hardware, unless some form of emulation or backward compatibility allows the software to run on new hardware. It is important to note that the term "legacy" refers to the state of a system before the strategic change. Legacy is a function of the change of a system. It is the result of the change of the environment. Without change, there would be no legacy. It is essential to realise that the legacy system is not useless. In most situations, the legacy system is important, valuable, and vital to the business organisations.

Process of Modeling the Legacy System

Structural Model

Cleaning Source Code

Reverse engineering a legacy system means going all the way back to the design stage from legacy source code. The original code may contain unstructured statements such as "GO TO" lines in programs written in COBOL, which cause the source code to become "spaghetti code" (Hutty & Spence, 1997). It is necessary to clean the original code and eliminate dead code.

Cleaning the code means obtaining an equivalent but different design with clearer and simpler semantics than the original code or representation. The representations are structured by the way to structure the code in order that each line of the source code is a meaningful fragment of a program specification (Pu & Yang, 2003b).

The process of cleaning the legacy software code can provide a clear description of the program in a readily understandable format, and thus form a solid base for further application. This involves migrating a legacy system onto a modern hardware or operating system platform, migrating an existing database to a relational database management system and converting a system to a modern programming language.

In the legacy system, there may be dead code that is useless to the execution of the tasks. Such dead code may have existed at the stage of the development. Or, with changes to the environment, especially due to improvements in the hardware, some methods of inputting or outputting data may have been modified, or some ways of storing data have been improved. Therefore the corresponding code becomes useless; on some occasions it may even result in failure of the system. Consequently that dead code must be recognised and removed from the legacy system (Pu & Yang, 2003b).

The legacy system will be improved over a series of increments, making functionality available to the user sooner than is possible with a big bang deployment strategy. The goal is to break up the migration to the future system into small manageable steps, where the objectives of each increment are well defined. Incremental plans are driven foremost by complexity and technical feasibility. It is critical to ensure that the functionality, reliability, and performance of the system are not diminished after the development of clean code has been completed through the removal of dead code. Incremental deployment also offers an opportunity for the organisation to gradually begin substitution of modernised components, easing the transition from legacy to modern technologies. At the completion of each increment, the percentage of the unaltered legacy system decreases while the percentage of cleaned code increases. Eventually, the legacy software code is completely cleaned.

Gathering Parameters and Operations

In the legacy system, all parameters are gathered together and important information about the main data structures is recorded in files or tables. Basically, a program is the set of all parameters and their characteristics and the operations on them. It is a semantic set within the specified operational environment. Therefore, the input parameters are served as the original raw data, the programming environment and the execution of program lines are regarded as the producing machines that change the characteristics of those parameters, and the output parameters are the outcomes that are produced during the execution of the whole software program. In some cases the input parameters are the same as the output, while in other cases they are different (Ben-Menachem & Marliss, 1997).

All parameters and associated operations are collected together. Those operations include systematic operations which may themselves include systematic calls.

Classifying Parameters and Operations

All the parameters and the operations on them are classified into several groups. Each group has one nucleus. Each group is related closely to each other to describe the common core. Although that core is not always obvious, each parameter and its related operations are part of the specifications of that nucleus in that group. In every group, each parameter depicts one part of the characteristics of that nucleus, such as its name, its age, its weight, its height, and its ID; and every operation is the change, assessment, or detection of those pieces of characteristics of that nucleus, such as increasing or decreasing its weight, confirming whether it is at that age, or determining whether it has another name.

It is possible that one operation may be involved in more than one group. The best way to deal with this is to separate it into several different operations, each of which concentrates on only one group.

Extracting the Classes

The nucleus that is contained in each group is regarded as a single class. Sometimes the name of the nucleus is not mentioned in the program, and it is then necessary to define a name for it. All parameters and operations in a group describe the attributes of the nucleus, the operations on the nucleus, and its relationships with other groups. These can be distilled as attributes and operations of the corresponding class (D'Souza & Wills, 1999).

The names given to classes, their attributes, and their operations should be domain-related. The domain is where the problem is allocated. All definitions and extractions must be based on domain knowledge (Yang, 1991; Pu, Millham, & Yang, 2003a). It should also be noted that an object is strictly not the same as its class. All objects with similar properties, including attributes and operations, are distilled into one class.

Defining Relationships of Classes

Classes that represent a legacy system have relationships between them. Each relationship presents one special aspect of the characteristics of the related classes.

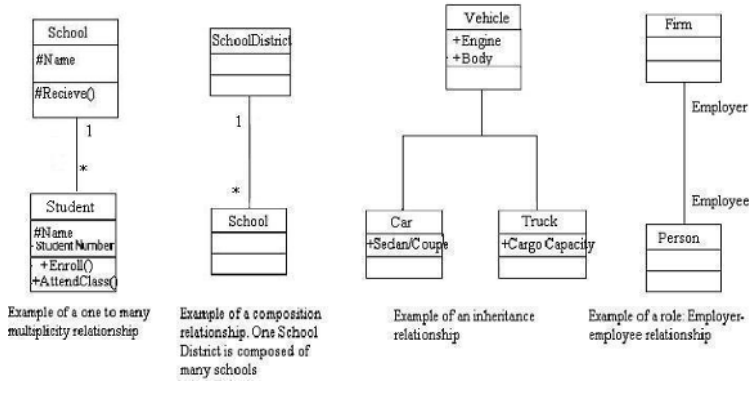
An association shows a relationship between two or more classes. Associations have several properties:

- A name that describes the association between the two classes. Association names are optional and need not be unique globally.
- A role at each end that identifies the function of each class with respect to the associations. Since the name of an association is optional, its roles are optional, too.
- A cardinality at each end that identifies the possible number of instances that can participate in the association.

Associations between the classes are domain-related. Initially, associations are the most important characteristics of the classes because they reveal more information about the application domain. Each association should be named as appropriate and roles assigned to each end.

It also is necessary to model generalisation relationships. Generalisation is used to eliminate redundancy from the analysis model. If two or more classes share attributes or behaviour, the similarities are consolidated into a superclass (Dorfman & Thayer, 1997). Other characteristics of classes include composition, multiplicity, inheritance, and roles. For example, in Figure 3, one school has many students; one school district is composed of many schools; all cars and trucks are vehicles; an employee works for an employer.

Figure 3. Examples of class composition, inheritance, roles, and multiplicity



Pretty Printing Class Diagrams

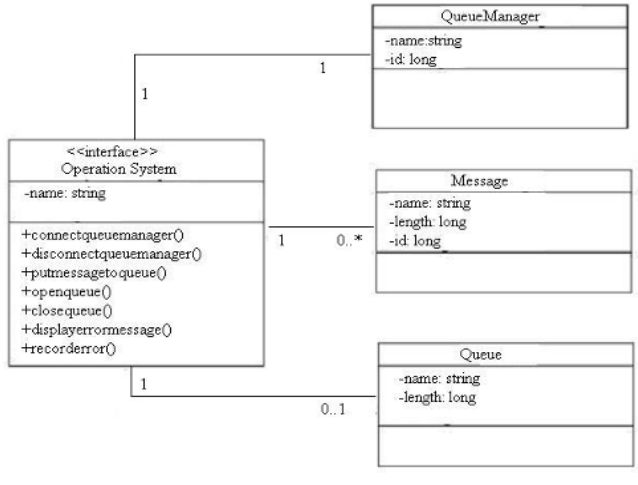
A class diagram is a diagram that shows a set of classes, interfaces, collaborations, and their relationships. Class diagrams are typically used to explore domain concepts in the form of a domain model, analyse requirements in the form of an analysis model, and depict the detailed design of object-oriented software. Class diagrams are essential to modeling the legacy system.

During the analysis development period, analysis models focus on the responsibilities of classes and not on their attributes or operations, while associations should be modeled as association classes. When association classes are modeled on analysis class diagrams, the dashed line of the association class is placed to the centre of the association and the association itself is not named. Only when a specific type of an attribute is a requirement of the system should it be described on an analysis class diagrams. Attributes and operations should be modeled on design class diagrams. During the design development period, four types of visibility of attributes or operations are defined: Public "+", Protected "#", Private "-", and Package "~". These are design details that define levels of access to the classes.

A class name is a singular noun based on common domain terminology. An attribute name is a domain-based noun and an operation name begins with a strong active verb. Graphically, a class is rendered as a rectangle with three compartments that contain name, attributes and operations, respectively. These three compartments are essential and necessary to a class even if one of the compartments is left blank. A class symbol may optionally contain other compartments, which should be labelled at the centre top. A compartment with an incomplete list should be marked by an ellipsis at the compartment end. Within each class compartment, attributes and operations should be listed in decreasing order of visibility, with static ones placed before instance ones.

If operation signatures are too long to fit the class symbol, only those object types that are passed as parameters to the operation are listed, in order to save space. Consistency

Figure 4. An example of a class diagram



of naming of attributes, operations, parameters, and of their ordering within classes is essential. Following language conventions, if the names of attributes and operations imply their stereotypes, the stereotypes should be omitted. Exceptions of operations can be indicated with a property string.

If two classes interact with each other, some kind of relationship may be needed between them. A transitory relationship is a dependency. It may not be necessary to model implied relationships or every single dependency.

In class diagrams, multiplicity between classes should always be shown. The indefinite multiplicity "*" can usually be replaced either by "1..*" or "0..*" and this should be done where possible. In some cases, attribute types can replace relationships. In normal modeling, the minima and maxima of multiplicities can be extended to "1..*" or even "0..*" to make class diagrams more flexible.

It is a common convention to centre the name of an association above an association path and use one or two descriptive words to name the association. The direction in which an association name should be read can be modeled with filled triangles; the common direction name is left-to-right. If multiple associations exist between two classes, class roles should be introduced to clarify the class diagram. Role names can be used to describe recursive associations that involve the same class at both ends. Only when collaboration can occur in both directions are the associations shown as bi-directional. Associations are inherited by implication and thus when changes occur to inheritance structures associations may need to be redrawn.

Inheritance models a generalisation relationship between a subclass (child) and a superclass (parent). It is usually modeled vertically with the subclass below its superclass, while other relationships are usually shown horizontally. A subclass inherits attributes and operations of a superclass. For inheritance to apply, one of the following sentences should make sense: "The subclass behaves in the same way as the superclass" or "The subclass behaves in a similar way to the superclass." If a subclass inherits only some characteristics (attributes or operations) but not others, an inheritance relationship between these two classes is not appropriate. A more formal test for inheritance is the Liskov Substitution Principle (Bennett, McRobb, & Farmer, 2002), which states in essence that it should be possible to treat a derived object (i.e., an instance of the subclass) as if it were the base object (i.e., an instance of the superclass).

Behavioural Model

Using Activity Diagrams to Understand and Model Behaviour

A UML activity diagram describes the dynamic aspect of a system. It is essentially a flowchart, showing flow of control from activity to activity. An activity can be defined as an ongoing non-atomic execution within a state machine (although activity diagrams do not necessarily model finite state machines). Activities ultimately result in some action, which is made up of executable atomic computations that result in a change in the state of the system or the return of a value. Actions encompass calling another operation,

sending a signal, creating or destroying an object, or some pure computation. An activity diagram models the sequential and concurrent steps in a computational process. It also can model the behaviour of an object as it moves from state to state at different points in the flow of control. Activity diagrams can be used to visualise, specify, construct, and document the dynamics of a society of objects, or they may be used to model the sequence of control of an operation (Booch, Rumbaugh, & Jacobson, 1999; Rumbaugh, Jacobson, & Booch, 1999; Larman, 1998).

Activity diagrams can model complex operations, business rules, business processes, and software processes. In addition to activity states, they can contain action states. An action state represents an an executable and atomic computation, such as an operation on an object, sending a signal to an object, or the creation or destruction of an object. An action state cannot be decomposed; events may occur, but the execution of the action state is not interrupted. The execution of an action state is generally considered to take insignificant time. In contrast, activities can be further decomposed with their activity being represented by other activity diagrams. Activities are not atomic, may be interrupted, and are considered to take some duration to complete. An action state is a special case of an activity state that cannot be further decomposed. An activity is thought of as a composite, whose flow of control is made up of other activities and actions. An activity can be extended to an activity diagram (Booch et al., 1999; Bennett et al., 2002; D'Souza & Wills, 1999).

Actions and activities are just special kinds of diagrams. An activity is semantically equivalent to expanding its activity graph until actions are represented. However, activities are important because they can help break down complex computations into parts. This is helpful when comprehending a legacy system that is large, complicated, and difficult to understand.

Activity diagrams are used to model the dynamic aspect of the legacy system (Systa, 2000). These dynamic aspects may involve the activity of any kind of abstraction in any view of a system's architecture. Activity diagrams model a workflow and the operation of a system. An activity diagram can be attached to any modeling element for the purpose of visualising, specifying, constructing, and documenting that element's behaviour.

Classifying Calls

A call stands for the procedure or function call in the programming language. The starting point in analysing the structure of the legacy system is to develop a call graph. Examining the calling structure of the legacy system can be used to identify program elements with minimal dependencies that can easily be migrated. Four different kinds of program elements are distinguished: root program elements that call other program elements but are not themselves called; leaf program elements that are called by other program elements but are not themselves called; node program elements that both call and are called by other program elements; and isolated program elements that neither call nor are called by other program elements.

Root program elements are typically programs that are invoked directly by the user or by some external process; otherwise, there would be no way to execute these programs. In

themselves, these program elements are not a good starting point for understanding the legacy system, since they call other program elements. The execution control goes back and forth among the elements of the legacy system.

Node program elements are even more difficult to migrate than root program elements. They are called by root program elements and in turn call leaf program elements. Thus, they share the difficulties of root program elements — that they call other elements — but must be migrated correctly from the legacy system so that the remainder of the legacy system can continue to function, permitting those node program elements to continue to be used by other program elements. Meanwhile, they share the difficulties of leaf program elements — that they are themselves called — and thus the flow of execution control goes up and down, and the executing procedures become more like "spaghetti code." They are the worst candidates for comprehending the legacy system.

Isolated program elements can be migrated easily. These elements could be converted in any given increment since their conversion neither increases nor decreases the number of elements that need to be developed from the legacy system.

Leaf program elements are the best candidates for the starting point in comprehending the legacy system. They do not call back to legacy source code, and although they require development, it is possible to minimise the number of these elements by transferring an entire subsystem in a single iteration.

Specifying the Behaviour of a Legacy System

Because a legacy system is typically large, unstructured, complicated, and old-fashioned, it is difficult to read and understand. Its specification must be realised from the source code. However, since the specification includes all the information of the source code, it is similarly complex in structure to the source code. Therefore the next work is to present the specifications in a graphical manner, which means extracting a high-level representation that is more understandable than the legacy system itself. This process will result in a great loss of detail, resulting in a description of the legacy system with UML activity diagrams. These describe the behavioural aspect of the legacy system (Yang, Luker, & Chu, 1997).

It is preferable to deal with the data structure, and to represent code as operations on data structures, rather than trying to recognise simple operations in the code and then attempting to limit the number of program variables involved in those operations. The specification of the system contains detailed information about interactions of variables and classes, detailed code structure, and a complete description of operations on variables.

Higher equivalent reasoning is essential. The data structures contained in the specifications can be eliminated to obtain a clearer description. Behavioural structures implement the dynamic aspect through internal constraints and hidden behaviour is represented explicitly. When a section of code implements a mathematical function, it is useful for these techniques to be applied as part of the object abstraction process, allowing a simpler description of a higher level behaviour or auxiliary behaviour to be obtained (Breuer & Lano, 1991).

Behavioural specifications consist of a set of top-level behaviour definitions, an explicit definition of the valid domain, and a set of lower level local specifications. After the code is improved, the specifications of the legacy system are generated entirely with a line-by-line translation mechanism. Although many new variables and executions of those variables are produced, and the first version of the specification from the legacy system is not very readable, the application of simple transformations presents an acceptable form. This is taken as the basis for the further transformations employed in equivalent reasoning.

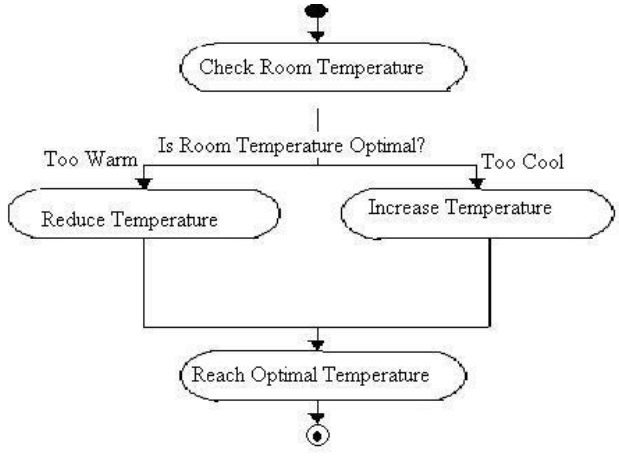
Specifications of the legacy system are presented with the purpose of describing the legacy software using real-world domain names, together any comments embedded in the programs — such as "student," "university," "bank," "This is to input data," "This is to print the file," or "This is to stop the program" — in order that the users of the system should clearly understand how it works. For large, complicated software with millions of lines of the programming language, it is essential to identify the detailed behaviour of the calls (Li & Yang, 2001).

Realising Activity Diagrams from a Legacy System

Activity diagrams permit a great deal of flexibility regarding what is modeled. A model is a communication device, and so requires an adequate level of detail to address the problem to be solved. Clarity and brevity are important to avoid visual overload, and a model should present key features, for example, of the control flows. Activity diagrams have an appropriate level of detail to describe system functionality.

When drawing an activity diagram, the starting point is usually located in the top-left corner of the activity diagram and an end point at the bottom-right corner. The start state is modeled with a filled-in circle, and the end state with a filled-in circle with a border around it. Every UML activity diagram has a starting point.

Figure 5. An example of an activity diagram that achieves an optimal room temperature



Each activity typically represents the invocation of an operation, a step in a business process, or an entire business process. Every activity (other than the start and end states) must have at least one transition into it and at least one transition out of it; otherwise one or more transitions must have been missed.

A decision point is modeled as a diamond. Each transition leaving a decision point must have a guard, and these are depicted using the format "[condition]." The condition must be true in order to traverse a transition. Guard conditions on transitions leaving a decision point help to describe the decision. The set of guards on all transitions from a decision point or an activity must be a complete and mutually exclusive set.

Activities that can occur in parallel are regarded as forks and joins. Each fork has one entry transition and should have a corresponding join with one exit transition (Ambler, 2002).

It is important to limit the level of complexity within each activity diagram. For example, if there are more than three possible paths (alternate or exceptional), additional activity diagrams should be used to promote understanding. Additional activity diagrams also can be used where the processing requires specific data elements.

Discovering Actors

It is important to identify the direct users of a legacy system, and this can only be done using knowledge from outside the system, since it involves human interaction and is also closely related to domain knowledge, neither of which is directly represented within the system. Candidate actors include humans who interact with the code, hardware that is external to the code, and other systems that interact with the system.

Software interacts with humans and also with other systems in the real world. All such interactors have a relationship with the legacy system. Users of the system perform tasks with it, and the code also can exchange information with other software, sending data to other systems, receiving data from them, or both.

When code is executed, it may exchange messages with other systems. It may send data to other systems and receive data from them, send control signals to other systems, receive control signals from them, or its execution may be initiated by other hardware systems. All these systems are regarded as interacting with the legacy system.

Each interactor has a different view of the legacy system. Each interactor is interested in a particular aspect of the legacy system. All interactors are considered to be candidate actors in the model of the legacy system that will be developed.

Each interactor should be regarded as a single actor. When there is a choice between a human actor and some hardware or software component, the human should be considered as the more important candidate actor. Only human actors can use the software directly or indirectly for their own purposes. Ideally, all users, maintainers, managers of the company, and even organisational sub-units should be considered as candidate actors.

The relative importance of each candidate actor should be carefully considered. While all candidate actors have importance in relation to the legacy system, some may be more important than others. Direct users are the most important of all candidate actors, since they use the legacy system and determine when it is executed. Operational results are reported directly to them, and they decide what should happen next. Next most important may be company managers, because they not only use the legacy system (directly or indirectly), but they also manage other direct users. For example, they may manage a business project based on the output of the legacy system. Both direct users and company managers use the legacy system for business purposes, and in some cases may even be the same people. Next most important are indirect users and maintainers, who in most cases do not use the legacy system for business purposes.

Other interactors besides humans are considered and may be modeled as actors, including the hardware that executes the legacy system and other systems that interact with it. This can describe the interaction, how the legacy system is executed by the hardware,and what messages are exchanged with other systems. Any system that invokes the legacy system is regarded as an actor.

Extracting Objects

The first step in object identification is to use static code analysis, where the degree of coupling and cohesion between variables and procedures are analysed. Procedures with high fan-in and low fan-out or with high fan-out and low fan-in are placed in separate classes. The reason for this separation is that procedures with high fan-in but low fan-out are typically logging modules, while procedures with high fan-out but low fan-in are typically controller modules that simply call other procedures. Eliminating these logging and controller modules (by placing them in separate classes from the rest of the analysis) helps to separate procedures whose only link with other procedures is functional cohesion. Functional cohesion occurs when two procedures perform a function together but do not share any other type of logical cohesion such as belonging to a common business unit. Otherwise, closely related variables and procedures are grouped into objects, with variables becoming object attributes and procedures becoming the object methods (Millham, 2002). Parameters are also collected together with their associated operations, including systematic operations such as systematic calls.

The validity of each object and its corresponding operations and attributes should be checked carefully for any overlap. If overlap between two objects occurs, the groups should be redefined. If an operation on one object contains an operation on another object, it is necessary to divide that operation into two parts, each of which focuses on only one object. If an inconsistency occurs in the code where the objects are accessed, it is necessary to restructure the code at a higher level in order to correct this type of logical flaws.

Attributes are properties of an individual object. When identifying attributes, only those properties that are relevant to the system should be considered. It is important not to confuse objects with attributes. An attribute consists only of a name that identifies it within its object, a brief description, and a type that describes the legal values it can take.

In the case of entity objects (that is, objects with little or no interesting behaviour), any property that needs to be stored by the system is a candidate attribute.

An association shows a relationship between two or more objects. Associations have several properties:

- A name that is used to describe the association between the two objects. Association names are optional and need not be unique globally.
- A role at each end that identifies the function of each object with respect to the association.
- A cardinality at each end that identifies the possible number of instances.

Associations between objects are important because they reveal information about the application domain; associations are domain-related. Each association should be named and roles assigned.

It is also necessary to identify generalisation relationships between objects. Generalisation is used to eliminate redundancy from the analysis model.

Aggregation is the typical whole/part relationship. This is operationally similar to an association with the exception that instances cannot have recursive aggregation relationships (i.e., a part cannot contain its whole).

Composition is exactly like aggregation except that the lifetime of the "part" is dependent on the "whole". This dependency may be direct or transitive. That is, the "whole" may take direct responsibility for creating or destroying the "part," or it may accept an already created part, and later pass it on to some other whole that assumes responsibility for it.

Refining Messages from Operations

A message from one object to another is an interaction between their respective lifelines. An object also can send a message to itself, that is, from its lifeline back to its own lifeline. Each message must be extracted from one or more operations. Similar operations that work together sequentially are collected together and presented as one message. For example, the three operations of sending the value of the day, the month, and the year to the date are extracted into one message named "display date." The source code for this is shown.

```
{
int day, month, year;
dim date;
{
day=today;
month=this-month;
year=this-year;
```

```
display date(day,month,year);
}
}
```

In the legacy system, some kinds of complex computation, especially mathematical formulae, are represented by messages in the application domain. Messages within the legacy system should be presented as sequence diagrams, and ordered according to their time and sequence of execution. For each message, the sending and receiving objects are recorded.

It is important to concentrate on critical operations to refine the messages of the legacy system. Not all operations are necessarily treated as messages, nor does one operation necessarily correspond to one message (Li & Yang, 2001).

Specifying Behaviour Using Sequence Diagrams

A sequence diagram is an interaction diagram that details how operations are carried out – what messages are sent and when. Sequence diagrams are organised according to time. Normally time proceeds down the page.

A sequence diagram has two dimensions:

- The vertical dimension represents time.
- The horisontal dimension represents object interaction.

The vertical line extending down from each object is called its lifeline. A lifeline represents an object's life during the interaction shown in the diagram. Messages are represented by horisontal solid arrows between the lifelines of the sender and receiver objects. Each message is labelled either with the name of the operation to be invoked or with the name of the signal. Argument values or argument expressions also may be represented. The arrow may optionally be labelled with a sequence number, and a message can also be prefixed with an iteration maker (*), which shows that the message is sent many times. Sequence diagrams are used to demonstrate the flow of control for a certain part of a program. They show how objects in the system interact based on messages sent and returned.

Layering is a common approach to the organisation of systems. As a result, it makes sense to layer sequence diagrams of the legacy system in a similar manner. This is based on layers of program calls, where the root program element is regarded as the first and most important sequence diagram. Other program elements are included in that diagram. Node program elements are presented before leaf and isolated program elements.

The primary actor of the legacy system is located at the top left side of the sequence diagram. Other actors follow in time and importance. Reactive actors (those that respond to outputs of the legacy system, but do not initiate interaction) are described at the top right side of sequence diagrams.

Message names are justified and aligned with the arrowhead. The message name is placed close to the recipient, and this makes sense since the receiver of the message implements the corresponding operation. The syntax of the legacy system's implementation language is utilised in naming messages, because this improves their understandability and readability.

In a sequence diagram, an object receives messages that invoke its operations in a time-ordered manner. Only when the operation that responds to one message has been executed, can the object respond to the next message in sequence. The timing of execution of messages by objects is clearly shown on a sequence diagram.

Legacy systems often use return values. When an object finishes processing a message, control returns to the sender of the message. This marks the end of the activation that corresponds to that message. The return of control is optionally marked by a dashed arrow from the bottom of the activation rectangle back to the lifeline of the object that sent the message giving rise to the activation. Activations and return messages need not be shown on a sequence diagram. When they are shown, they also may optionally carry a label that indicates the return value. When the next part of a sequence diagram refers to the return, it is necessary to model the return values. Otherwise, they are omitted for the sake of clarity and simplicity. This is a common modeling convention because it helps to visually determine the volume of message flows to a given object, and thus to judge the potential coupling that object has with other objects, which is often an important consideration for refactoring the design.

Comprehending the Legacy System with Collaboration Diagrams

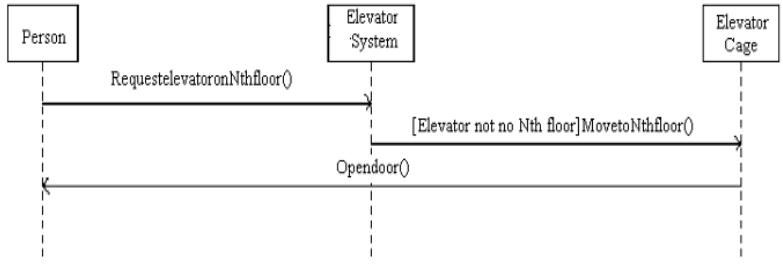
A collaboration diagram is exactly equivalent to a sequence diagram; therefore, it is produced following the same steps as for the realisation of a sequence diagram.

In a collaboration diagram, object roles are represented as boxes and association roles are shown as solid lines that follow the association paths between the objects.

If the system being modeled has a layered architecture, it makes sense to layer collaboration diagrams in the same way as sequence diagrams.

As for sequence diagrams, message names are justified and aligned with the arrowhead. The receiver of the message implements the corresponding operations and it makes sense

Figure 6. An example of a sequence diagram



that the message name is close to that message recipient. The syntax of the implementation language of the legacy system is utilised in naming the messages. That improves the understandability and readability.

In a collaboration diagram, an object receives messages and invokes the operation of time ordering. Only when one message has been executed, can the next message then be performed in the time dimension. So the time periods of executing the messages by objects are clearly shown with the numbering of those messages on collaboration diagrams.

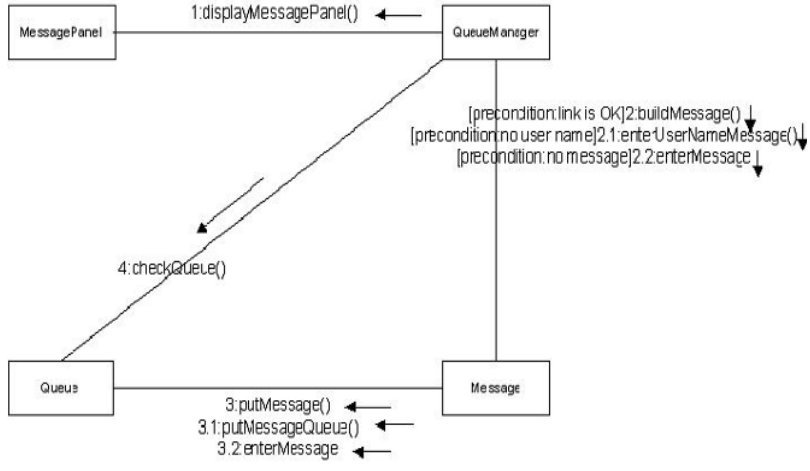
Since collaboration diagrams are logically equivalent to sequence diagrams, similar considerations apply regarding the omission of return values to improve clarity and readability. When they are indicated, this is shown with an arrow labelled to indicate the return value. Activations also are optional on a collaboration diagram; when they are shown, they are represented by an arrow alongside the association path.

Modeling the Legacy System Using Statecharts

Statecharts differ from activity diagrams in that they can only model those elements of the legacy system whose behaviour can be represented formally as a finite state machine. In order to determine events, a run-time environment for the legacy system must be created and then, with a comprehensive set of test data, the event/condition sequence output by the legacy system (in response to the test data) is recorded and analysed.

Systa and Koskimies (1997) describe a process of extracting state diagrams from a legacy system by first creating an event trace diagram that models the interaction of a set of objects and actors during a specific usage of a system, and then uses this event trace diagram to create a state diagram. Given a comprehensive set of test cases that provides

Figure 7. An example of a collaboration diagram for sending messages



coverage of all possible behaviours of a system, the legacy system is run using these test cases as input.

The running system is monitored for event and condition sequences that are produced by objects of that system. This event/condition sequence is sent to a tool, SCED, which constructs a scenario diagram that models the interactions of a set of objects implied by the event/condition sequence. SCED then is used to synthesise the general behaviour of an object as a state diagram, given a set of scenario diagrams in which the object participates (Systa & Koskimies, 1997; Booch et al., 1999).

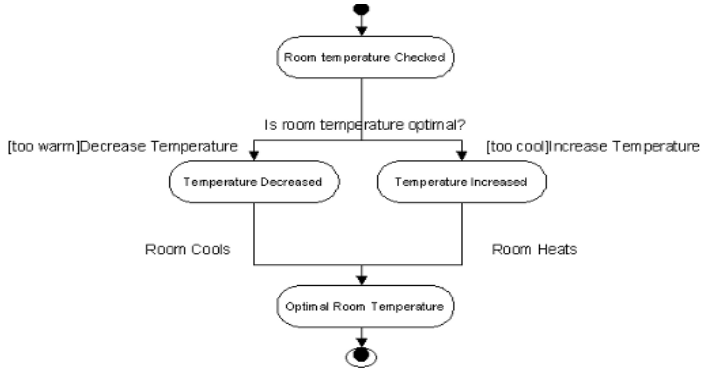
Domain Model

Defining Use Cases

A use case represents a task that a user of the legacy system (represented by a selected actor) wants the system to do (Howe, 2002). At a more abstract level, the main purpose of all users of the legacy system, including direct and indirect end users and company managers, is to achieve success for the business in its wider environment. Other persons, who are not themselves users, may also gain some benefit from the operation of the legacy system. Other systems and the hardware on which it runs provide support for it. As a user-centred analysis technique, the purpose of a use case is to yield a result of measurable value in response to an actor's request (in most cases, the actor can be considered to be acting in a wider commercial environment, and it is within this wider context that the value is defined).

The legacy system has responsibility for accomplishing the expected tasks for each actor. It is thus important to define what that actor wants the system to do. The starting point for this may be just a vague notion of the main task that the actor wants the legacy software to do, and this may contain ambiguities, inconsistencies, confusion, and incompleteness (Yang, 2000).

Figure 8. An example of a statechart that achieves an optimal room temperature



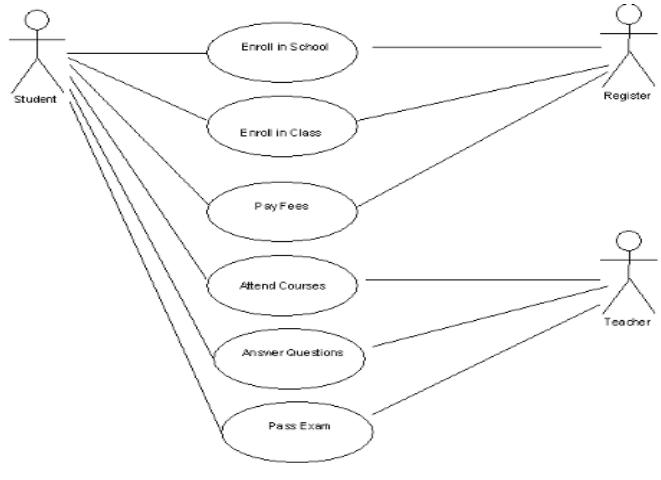
The demands of each actor must be defined and analysed in order to resolve conflicts and remove ambiguities. To do this, it is necessary to collect all requirements for each actor, identify those functions of the legacy system which contribute to that actor's purpose, along with any quality attributes such as performance, safety, portability, environmental questions, and so on, to form a viewpoint that represents all parts of the system which affect that actor.

Each use case executes a function of the legacy system that is termed the "usage." The main tasks that the legacy system is able to perform together represent its functionality. The choice of actors is closely related to the identification of these tasks.

An actor may use the legacy system for more than one purpose. Because the use case is utilised to present the main tasks that the actor would like to perform, detailed and common performance of the program should be ignored. For example, an actor may require the software to accept new input, carry out some arithmetical calculation, put stored data in some particular sequence according to a supplied parameter, output the result to a file, and then print it. These requirements do not need to be presented as, or in, a single use case. For a legacy system that has more than one actor, each actor's requirements should be prioritised in order of importance. Meanwhile, a collection of associated requirements may be conflicting and ambiguous. If the specification consists of requirements that are simple and not too closely related to each other, it will work well. But if, during the specification of the legacy system, it is found that the requirements are repeated or overlap with each other, it will be necessary to understand the overall tasks of the legacy system first, and then to distinguish in detail which among them are routine and/or frequently executed.

Each task that an actor wants the system to accomplish can be regarded as a candidate use case. Thus, the various purposes that actors have in requesting the legacy system to carry out some task can generate candidate use cases for that actor.

Figure 9. An example of use cases involving a student, registrar, and teacher in a university



Each task related to a specific actor within a large, complex legacy system, then, represents a development from a completely detailed use case for that actor (Reed, 1998). Thus, to identify use cases correctly requires an understanding of the external users' point of view and is handled in the business domain.

Once one use case has been identified and modeled for a selected actor, other tasks related to the same actor can be identified, extracted, and modeled as use cases for the same actor. Thus, all use cases with a relationship to one actor are extracted and defined together.

Systematic calls within the legacy system also are regarded as use cases. Such calls invoke the operating system to perform specific tasks. They send parameters to the operational environment and use feedback from the environment to achieve a result. How the operational environment performs its tasks is not shown in any detail. But procedures executed within the legacy system are distinct from systematic calls and they also are modeled as being initiated by use cases. They represent the way in which the tasks are performed and include subprograms and functions.

For each use case modeled, it is important to define a basic course. This is a main task for the corresponding actor. The main tasks of the legacy system should be modeled first.

Use cases are categorised as primary if they are main functions of the system, and secondary if they are secondary to the main purpose of the system (e.g., they are rarely executed). A primary use case is represented as a brief description of the main processes used to accomplish the corresponding system functions (Graham, 2000). Each major process corresponds to a primary use case of the modeled system.

Once the basic courses of the primary use cases have been defined, other tasks that seldom occur, or that are infrequently followed alternate courses of a primary use case, are defined as secondary use cases.

Systematic calls that are secondary tasks, or that are seldom executed, correspond to secondary use cases. Similarly, procedures that are secondary to the main purpose of the system, or that are not frequently executed, correspond to secondary use cases.

Use cases are documented by a use case name, the initiating actor (and any other communicating actors), the type of use case, and a description of the processes invoked. The syntax of a primary use case is as follows:

- Name: system action
- Actors: Actor1 (Initiator), Actor2, other systems,
- Type: primary
- Description: the use case begins when Actor1 interacts with the system.

Secondary use cases are similar apart from their type.

Use case descriptions should be compared to each other, since it is important to differentiate the goals of one use case from another. Any overlap, incompleteness, inconsistency or ambiguity should be removed. Any commonality between different use

cases can be extracted into separate use cases that are related by either an <extends> or an <includes> relationship. This is reported to be the most efficient way of finding use cases related in this way (Rubin, 1998). <Extends> and <includes> relationships are discussed further in the next section.

Finally, one use case may be invoked by more than one actor and, similarly, one procedure may be called by more than one other procedure in the legacy system. Thus at this stage, it also is important to record the interaction between actors and use cases, and to decide which use cases can be invoked by only one actor and which by more than one actor (Graham, 2001).

Modeling the Legacy System with Use Case Diagrams

It is recommended that use case diagrams should be drawn in sequence, according to their relative importance for the system and their sequence of execution in time (Ambler, 2002).

Each use case is named with a phrase that begins with a strong verb and this should be consistent with domain terminology.

An actor represents a coherent set of roles that a user of a use case can play, and not necessarily an individual job or position within the organisation. An actor's name should be a singular, domain-related noun. Actors are drawn outside the boundary of a use case diagram (since they are regarded as distinct from the system that is modeled). An actor regarded as a primary actor is placed in the top-left corner of the diagram.

Actors can interact with one or more use cases but cannot interact with other actors. No relationship between actors can be represented in a use case diagram (except for generalisation), even when in reality the individuals or systems represented by actors do interact with each other. Any interactions that do occur between actors are recorded in text notes, and are not strictly part of the use case diagram itself. If there is a system actor, this is stereotyped as <system> (Heywood, 2002).

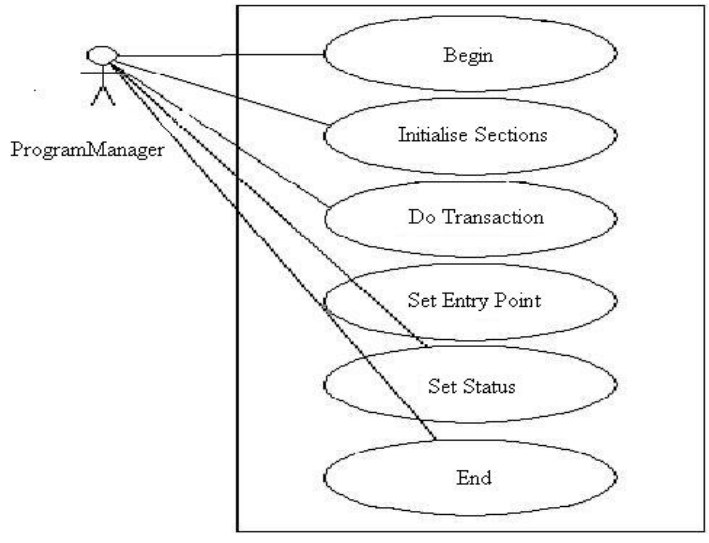
It is recommended for the sake of clarity and simplicity to avoid more than four levels of use case associations. Included use cases are placed to the right of the invoking use case, while extending and inheriting use cases are normally placed below the parent use case.

If generalisation between actors is defined, this should be checked carefully for consistency between parent and child actors. Similarly, names, meanings, and domains of actors and use cases should be checked for consistency, as this is essential for the reader to have a clear understanding of the models.

The system boundary of the legacy system indicates the scope of the model. A use case diagram only shows interactions of actors with the code of the legacy system. It should not include the boundaries of other software and programs.

Based on the identification of all use cases for a selected actor, relationships among use cases also can be identified. This principally applies to the discovery and modeling of <extends> and <includes> relationships between use cases.

Figure 10. An example of a use case diagram



Each actor has an association with at least one use case, which represents the ability of that actor (i.e., a user of the legacy system in the role designated by the actor) to initiate the use case or to transfer messages to or from the use case. An <extends> relationship (Booch et al., 1999) between two use cases signifies that the base use case (the one that is extended) implicitly incorporates the behaviour of another use case (the extending use case) at a specified point in its behaviour.

The base use case may execute alone, but also, under certain conditions, its behaviour may be extended by the behaviour of another use case. An <extends> relationship is a way to model a part of the use case behaviour that a user regards as optional system behaviour. An <includes> relationship between two use cases signifies that the base use case (the including use case) explicitly incorporates the behaviour of another use case (the one that is included) at a specified point in its behaviour (Larman, 1998).

Commonality between the definitions of two actors may be modeled using the generalisation concept (in exactly the same way that commonality between two classes can be abstracted as a generalised superclass). A generalisation relationship between actors signifies that the child actor inherits both the behaviour (in other words, the use cases) and the semantics of the parent actor.

There are many tools that help to draw use case diagrams. These include Rational Rose, UMLet, Artisan Real-time Modeler and Real-time Studio Professional, Atos Origin Delphia Object Modeler, Documentator, Logic Explorers Code Logic, and MasterCraft Component Modeler.

A Reverse Enginerering Tool, TAGDUR

Many software tools support reverse engineering, forward engineering, and reengineering. According to Müller (2004), reengineering tools may be categorised in several different ways according to their function. Some tools function as analysis tools by extracting artefacts, such as call graphs and metrics, from the legacy system. Other tools function as an understanding environment which parses the legacy system and stores the extracted software artefact in a repository for querying, behavioural pattern matching, and abstract representation. Still other tools offer an integrated forward and reverse engineering environment that incorporates both analysis and understanding tools with the ability for code generation. Furthermore, tools can be designed for scale, extensibility, or applicability and can be integrated along control, data, and presentation lines (Müller, 2004).

Integrisoft's Hindsight tool is designed for program understanding with the ability to provide documentation of the program's control flow, data structures, test coverage, and complexity. Viasoft's Existing Program Workbench (EPW) tool is a parsing engine that also provides documentation of the program's control and data flow. EPW decomposes a large COBOL program into smaller, more manageable units through program slicing and code extraction. Telelogic's Logiscope is a program analysis tool. IDE's StP/SE and StP/RevC is an integrated forward and reverse engineering toolset for C. McCabe's Visual Reengineering Toolset analyses systems written in multiple programming languages such as C, COBOL, and Fortran. Reasoning's Software Refinery generates tools for reverse engineering. This tool has the features of executable program specifications and rule-based program transformations.

Modeling a system into UML diagrams during the reverse engineering process poses some particular problems. UML was designed to model object-oriented systems; legacy systems which are the target of most reverse engineering efforts, tend to be procedural rather than object-oriented. Furthermore, these legacy systems tend to be designed to operate in a strictly sequential manner and to respond to procedural invocations rather than events. In order to enable this type of legacy system to be modeled in UML, it is important to first transform the original legacy system from a procedurally structured and strictly sequential-operating design to an object-oriented, event-driven system.

TAGDUR is a forward and reverse engineering toolset which, combined with Fermat, has the ability of program analysis, dead code elimination, rule-based program analysis and transformation, and code generation. TAGDUR was designed to automate the restructuring and re-documentation of batch-oriented legacy systems using source code as the only available software artefact.

TAGDUR also documents the transformed system via a series of UML diagrams. The UML diagrams produced by TAGDUR are written in a WSL notation that maps syntactically to the corresponding diagram structures in the UML 1.5 Specification (OMG, 2004). By incorporating the latest fully available version of UML, version 1.5, into its WSL diagrammatic notation, TAGDUR ensures that its diagrammatic notation correspond to UML's most recent specification. WSL is extended to represent high-level constructs such as UML's classes, associations, and activities (Yang & Ward, 2003). By keeping

this diagrammatic notation within WSL, TAGDUR maintains the advantages of the WSL representation of the system, notably tool independence, but because the WSL and UML diagrammatic notation are syntactically similar, conversion from this WSL notation into any UML exchange format, such as UXF, and hence, their importation into many UML graphical tools is easily accomplished. Furthermore, through WSL (Yang & Ward, 2003), there is a direct relation between the original source code, the model of the restructured system, and the reengineered target system.

Transforming Procedural Legacy Systems into Object-Oriented Systems

Before the transformation, we first convert the legacy system from its original programming language into WSL using a set of conversion rules particular to that original programming language which were formulated using Martin Ward's paper, *The Syntax and Semantics of the Wide Spectrum Language,* which defined the basis of the Wide Spectrum Language. Wide Spectrum Language (WSL) is a mathematical, intermediate language (Ward, 1992). WSL is called wide spectrum because this language represents both high- and low-level constructs. Consequently, WSL is ideally suited to represent all types of programming languages. Furthermore, this conversion from original programming language gives the converted system implementation independence such that any transformation of this program is independent of the programming language that the system was originally developed in.

WSL was chosen for an intermediate language for several reasons. WSL is programming and platform independent. Consequently, the transformations and modeling that TAGDUR performs on a WSL-represented system could be performed regardless of whether the original legacy system was in COBOL or C. The original legacy systems need only to be converted into WSL first.

WSL has other advantages as well. WSL has excellent tool support, in the FermaT transformation system which allows transformations and code simplification to be carried out automatically. It has the capability of enabling proof-of-correctness testing. WSL is programming and platform independent. WSL also was specifically designed to be easy to analyse and transform (Yang & Ward, 2003).

This transformation process involves three main steps. The first step is object identification using static code analysis where the degree of coupling and cohesion between variables and procedures are analysed. Procedures with high fan-in and low fan-out or with high fan-out and low fan-in are placed in separate classes. The reason for this separation is that procedures with high fan-in but low fan-out are typically logging modules, while procedures with high fan-out but low fan-in are typically controller modules that simply call other procedures. Eliminating these logging and controller modules, by placing them into separate classes, from the rest of the analysis helps separate procedures whose only link with other procedures is functional cohesion. Functional cohesion is when two procedures perform a function together; these two procedures might not necessarily share any other type of logical cohesion such as belonging to a common business unit. Otherwise, closely related variables and proce-

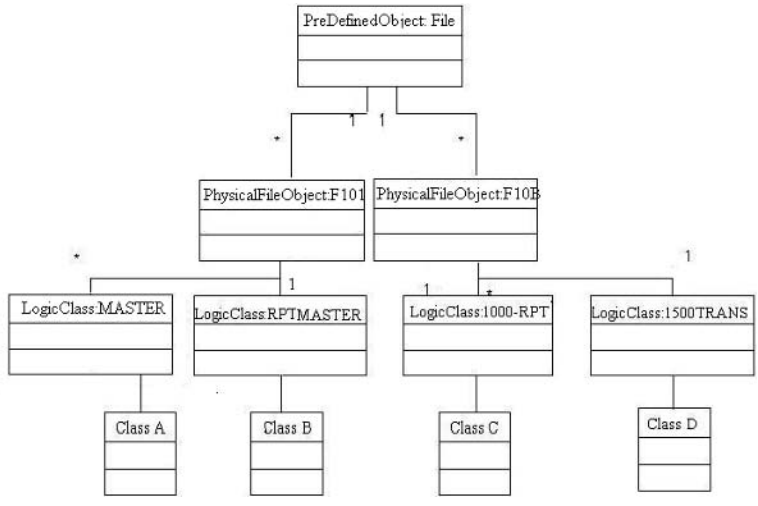
dures are grouped into objects with the variables becoming the object's attributes and procedures becoming the object's methods (Millham, 2002). The next step of this transformation process is to analyse two or more normally sequential units (whether these units are procedures, program blocks, or individual code lines) for dependencies. If there are no dependencies, the units that are being evaluated are deemed to be able to execute independently; otherwise, if there are dependencies, the units being evaluated are deemed to execute sequentially only. A dependency is defined as a simultaneous read and write operation of the same shared variable by two or more granular units of execution. Simultaneous reads and writes of the same shared variable results in an inconsistent state; for example, the value read by task A may be different depending on whether parallel-executing task B updates this shared variable before or after the variable is read by task A. Because such an inconsistent state cannot be allowed, these tasks must execute in their original sequential order rather than be allowed to execute independently (Millham, Yang, & Ward, 2003a).

The third step in this process is to identify possible events in the system and to model these events as asynchronous or synchronous events in UML. Although events occur outside the application domain, events such as user input, in many batch-oriented legacy systems, most events occur within the application domain. In other words, in batch-oriented legacy systems, the only external event is the arrival of batch input. All other events occur within the application domain. The latter types of events include input/output operations, procedure invocations, and system interrupts and exceptions. This transformation step consists of parsing the source code in order to identify these possible events. These events, once identified, are then analysed in order to determine if any dependency occurs between the code line where the event occurs and code lines immediately successive to this code line. If a dependency exists, which means that a system must wait for the event handler invoked by the occurrence of the event to complete its execution before resuming its normal post-event execution, the event is deemed to be synchronous. Otherwise, if no dependency exists, the event is deemed to be asynchronous. It is necessary to determine if each event is asynchronous or synchronous before they can be properly depicted as such in various UML diagrams which model these events.

Creating UML Diagrams Using TAGDUR

In many legacy systems, finding system artefacts to use as a basis for modeling UML diagrams of the system is difficult. Any documentation of the system often does not exist. The original developers and end-users, who would be most knowledgeable about the design of the system, have long since left the organisation. Often these systems have been left in light maintenance mode for many years; consequently, current maintainers and end-users have a minimal knowledge of the system. Because the source code, along with the associated data files, are the only available system artefacts, any UML diagrams that are generated to model this system must be based primarily on source code.

Figure 11. An example of a class diagram



Class Diagrams

Class diagrams represent the static structure of the COBOL legacy system. Class diagrams convey information about classes used in the system such as their properties, their interfaces, and how these classes interact with one another. Associations are relationships between instances of two classes. The attributes of the class describe its characteristics. For example, if one class is a student, its attributes may include the age, the sex, the class, the name, and the ID.

After our reverse engineering process transforms the legacy system from its procedural structure to that of an object-oriented one, our tool extracts the class diagram from the transformed system. Class definitions from the system are modeled as classes in the UML class diagram; variables encapsulated within a class become class attributes, and procedures associated with a class become methods in the UML class diagram.

Classes in this system are grouped into UML packages. A package, in this legacy system example, corresponds to the original COBOL copybook. The assumption is that the original programmers divided up the system into modules, in this case COBOL copybooks, according to some logical criteria. This logical modularisation of the system is preserved in the form of packages in UML diagrams. These packages with their classes also can be considered frameworks of classes with each framework retaining the logical partitioning criteria that divided the original system.

Classes that have been identified within the legacy system may be aggregated into a super-class hierarchy using a number of criteria. Highly coupled groups of classes may be grouped into a super-class hierarchical structure (Pu & Yang, 2003b).

Since this legacy system makes much use of logical file names rather than physical file names, I/O operations may be modeled as interactions between the class whose method invokes the I/O operation and a class deemed to represent the logical file. Because many of the latter class may access the same physical file, these logical filename classes may be deemed to be subclasses of the physical filename class. In turn, the physical filename class is, in turn, deemed to be a subclass of the File class itself.

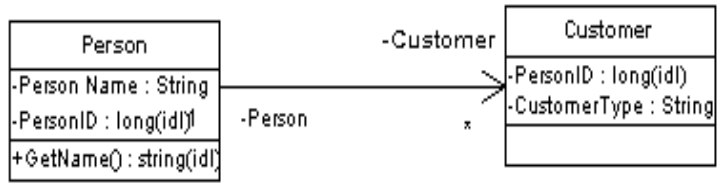
In this example, four classes, whose methods each invoke an I/O operation, interact with a logical filename class. These logical filename classes, in turn, are grouped, as part of a compositional relationship, with their physical filename classes, F101 and F102. These physical filename classes are then grouped, as part of a compositional relationship, with the predefined super-class, File.

Our tool models accesses between the classes of other attributes or methods as static associations between the classes. Each end of an association contains a multiplicity; the multiplicity of an association end is the number of possible instances of the class being associated with a single instance of the other end. Depending on the ratio of classes accessing the attributes/method to the classes accessed, the multiplicity of these association ends may be modeled as many-to-one, one-to-one, one-to-many, and many-to-many. For example, if an instance of Class A accesses methods and attributes of multiple instances Class B, this association end would be modeled as many-to-one. If an instance of Class A accesses the attributes and methods of just one instance of Class B, then this association end would be modeled as one-to-one multiplicity.

The information gained during the transformation process of this system is used in modeling these multiplicities. During the object identification process, TAGDUR constructs two matrices: one matrix is a procedural usage grid which records the number of times procedure A is called by procedure B; the other matrix is a variable usage grid which records the number of times variable A is accessed within procedure B. These matrixes are used during the object identification process where highly coupled procedures and variables are grouped into classes. These matrices also are used when modeling UML diagrams.

Variables or procedures that form the attributes and operations of one class, class A, but that are accessed by procedures that form operations of another class, class B, are modeled as an association between classes A and B. These two usage matrices are used when modeled the multiplicity of these associations.

Figure 12. An example of a class diagram with two classes and a one-to-many association between them



After the class aggregation process is complete, both procedural and variable usage matrices are used to determine the number of times that the variables and procedures of instances, or objects, of class A are accessed by procedures of an instance of class B; this usage forms the multiplicity relationship between classes A and B. For example, if the procedures of an object of class B access the variables and procedures of 10 objects of class A, the association between classes A and B is modeled as an association of 1:n multiplicity (Millham & Yang, 2003b).

Example of UML to WSL Notation Mapping

A small example of a class diagram is given. This diagram consists of two classes, Person and Customer, and has an association of one-to-many between them because one Person may represent several Customers to a company.

Each package consists of a set of classes within a diagram. Hence, the WSL notation of a package, defined as a structure, would consist of both a package name and a class name list. The WSL notation of class, also defined as a structure, consists of a class name, a visibility name, an attribute list, and an operations list.

An attribute list is a list of ordered pairs in the format (VisibilityName: AttributeName). VisibilityName can be one of the following visibility options: + public, # protected, - private, and ~package.

An operations list consists of a list in the format <VisibilityName> <OperationsName>(<ParameterList): <ReturnTypeExpression><PropertyString>. <OperationsName> is the name of the Operations; <ReturnTypeExpression> is an optional expression that describes what an operation returns. <Property String> indicates property values that apply to the element. In the case of Operations, these property values may include <Query> which, if true, indicates an operation that does not modify the system state and <Concurrency> which has the following values: sequential, concurrent, or guarded (the operation's invocation is governed by a guard condition).

The parameter list is in the format: <KindName><ParameterName>:<Type-Expression> = <Default_Value>. <KindName> indicates the parameter direction such as in, out, or inout. <Type-Expression> is the data type of the parameter. The Association consists of a structure including the fields of AssociationName, MultiplicityName, AssociationEndName1, and AssociationEndName2. The AssociationEnd structure has the fields of Ordering (whose value can be ordered or unordered), Navigability (whether an association is navigable or not), AggregationIndicator (if yes, indicates that the association is an aggregation association between classes), RoleName, Changeability (if this association is changeable or not), and VisibilityName. Multiplicity may be a one-to-one [1..1], one-to-many [1..*], zero-to-one[0..1], zero-to-many [0..*], and many-to-many [*..*] (OMG, 2004). Because WSL is typeless, each WSL structure has an additional field ElementType which indicates what UML element this structure represents.

In the previous example, the WSL notation would be as follows:

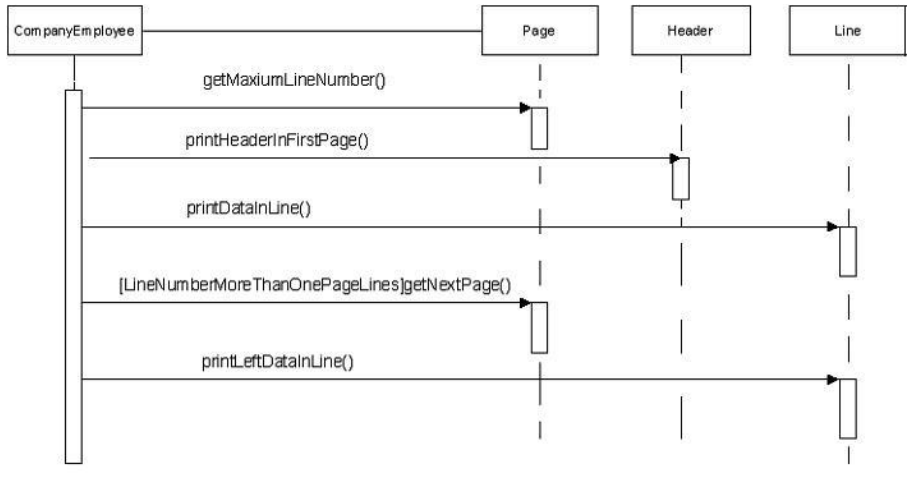
```
Var Struct ClassPackage
    Begin
    Var ElementType = Package
    Var PackageName = Package
    Var ClassList = Person, Customer
    Var AssociationList = PersonCustomerAssociation
    End

Var Struct Person
    Begin
    Var ElementType = Class
    Var AttributeList = {Private, PersonID}, {Private, PersonName}
    Var OperationsList = Public GetName():String
    End

Var Struct Customer
    Begin
    Var ElementType = Class
    Var AttributeList = {Private, PersonID}, {Private, CustomerType}
    End

Var Struct PersonCustomerAssociation
    Begin
    Var ElementType = Association
    Var MultiplicityName = [1..*]
    Var AssociationEndName1 = PersonAssociationEnd
    Var AssociationEndName2 = CustomerAssociationEnd
    End

Var Struct PersonAssociationEnd
    Begin
    Var ElementType = AssociationEnd
    Var AssociationEndName = Person // name of linked class object
    Var Ordering = unordered
    Var Navigability = yes
    Var AggregationIndicator = none
```

Var RoleName = 'No Role'
Var Changeability = Yes
Var Visibility = Public
Var Multiplicity = [1..*]
End

Var Struct CustomerAssociationEnd
Begin
Var ElementType = AssociationEnd
Var AssociationEndName = Customer // name of linked class object
Var Ordering = unordered
Var Navigability = yes
Var AggregationIndicator = none
Var RoleName = 'No Role'
Var Changeability = Yes
Var Visibility = Public
Var Multiplicity = [1..*]
End

The purpose of this UML to WSL structure mapping is twofold: One is to extend WSL to represent high-level abstract modeling concepts and elements; the other is to make this mapping UML-compliant such that these structures can easily be exported, via XML, to various UML tools.

Figure 13. An example of a sequence diagram



Sequence Diagrams

A sequence diagram is an interaction diagram that details how operations are carried out — what messages are sent and when. Sequence diagrams are organised according to time with time progressing as you go down the page. The objects involved in the operation are listed from left to right, according to when they take part in the message sequence.

Sequence diagrams are modeled as one line; this line contains all the objects in the system with each object in its own swimlane. Because all objects have a global scope, and variables in COBOL are globally scoped, these objects are shown as being immediately activated after their declaration in the swimlane. Messages are depicted with their origin at the source object and their endpoint being the target object. These messages are depicted as synchronous or asynchronous.

Exceptions are depicted as messages originating in the object where the exception occurred and ending at System object, which handles exceptions. Similarly, system interrupts are depicted as messages originating in the object where the interrupt occurred and ending at System object, which handles interrupts.

Procedure calls between objects are depicted as messages between the source, or caller, object and the target, or callee, object. Exceptions and interrupts are modeled as messages between the object where the interrupt/exception occurred, the source object, and the System object, the target object. Procedure I/O calls, such as Put or Fetch statements, are modeled as messages between the source object, where the procedure I/O calls is invoked, and the File object, the target object.

Depending on whether immediately successive tasks are dependent on the result of these messages as determined during the identification of independent process, these messages are depicted as asynchronous or synchronous in the sequence diagrams.

The determination of independent tasks' process assigns sequence numbers at the procedural and individual codeline granularity. The order of sequence number indicates the order of execution such as the task(s) with the lowest sequence number being executed first followed by the tasks with the next lowest sequence number and so on. Tasks with the same sequence number may be executed in parallel.

Depending on whether the task immediately succeeding the message task is dependent on the message finish execution first, the message is modeled as synchronous, if such a dependency exists, or as asynchronous, if no such dependency exists.

Each message depicted in the sequence diagram is given a sequence number in the format: <procedure task sequence number>.<individual codeline task sequence number>. The procedure task sequence number is the sequence number of the procedure where the message is invoked and individual codeline task sequence is the sequence of the codeline where the message is invoked. The sequence indicates the order of execution in ascending order; messages with the same sequence number may be executed in parallel.

We model sequence diagrams as message passing between objects of the system (Millham & Yang, 2003b).

Component Diagrams

Component diagrams model software components and their relationships within the implementation environment. These components may be simple files or dynamic libraries. Relationships between components are modeled as dependency relationships; a relationship between two components is identified when one component offers services to other components. Generally, these components represent compilation dependencies. Several of these components may be grouped into packages, or subsystems, according to some logical criteria.

TAGDUR models component diagrams in its re-engineering process. Often the legacy system consists of a main program file that calls, or loads, several program sub-files which, in turn, load other program sub-files. This sub-file call hierarchy is first identified by parsing the source code of legacy file in order to identify which sub-files are loaded from which files; this call graph is modeled as a component diagram (Millham, Pu, & Yang, 2004).

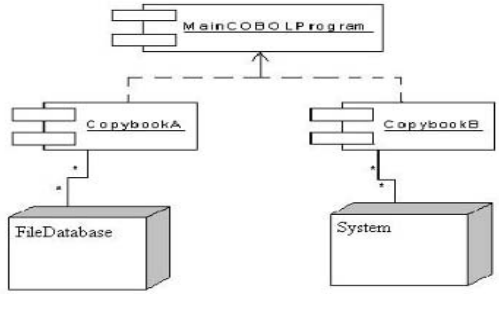
Deployment Diagrams

Deployment diagrams show the physical configurations of software and hardware. Our tool parses the source code to identify any relationships between the classes or packages that contain the source code and any external entities that this source code refers to. For example, a WSL statement in the source code might access a Terminal device. Parsing by this tool will reveal the relationship between the class containing this WSL statement and the Terminal object. This relationship is then depicted in the Deployment diagram.

Deployment diagrams are derived by parsing source code to determine the possible relationship between the class containing the source code being parsed and external entities such as a file system and its physical files, peripheral devices such as a printer, or user interfaces.

Physical devices, such as printers, are modeled as nodes in deployment diagrams. Program modules, such as the original COBOL Copybooks, are modeled as component

Figure 14. An example of a deployment diagram



instances. These modules are composite parts of the legacy system (original source code programs) (Millham et al. 2004).

Activity Diagrams

Activity diagrams describe the internal behaviour of a class method as a sequence of steps. These sequence of steps model the dynamic, or behavioural, view of a system in contrast to class diagrams, which model the static, or structural, view of the system.

An activity in UML represents a step in the execution of a business process. Activities are linked by connections, called transitions, which connect an activity to its next activity. The transitions between activities may be guarded by mutually exclusive Boolean conditions. These conditions determine which control flow(s) and activities are selected.

Activity diagrams may contain action states. Action states are states that model atomic actions or operations. Activity diagrams also may contain events.

Activity diagrams can be partitioned into object swimlanes that determine where an activity is placed in the swimlane of the object where the activity occurs.

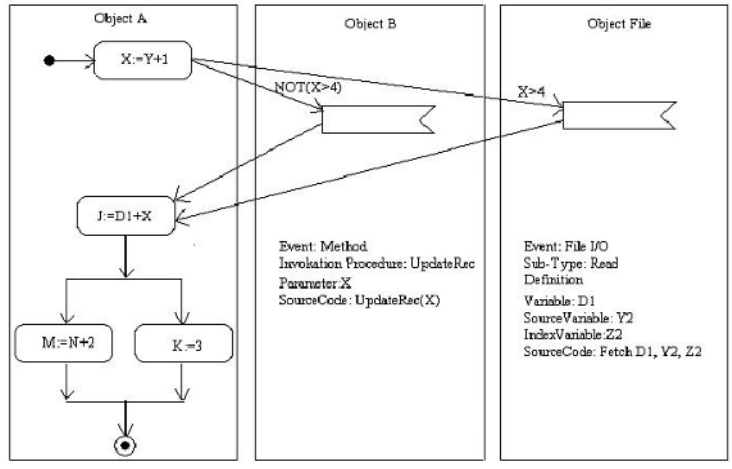
Tasks that have been determined to be able to execute in parallel by the independent task evaluation step of the transformation process are modeled as parallel activities and flows in the activity diagram, while tasks that have been determined to be able to execute sequentially only are modeled as sequential activities and flows. In activity diagrams, synchronisation bars are used to synchronise the divergence of sequential activities into parallel tasks or the merging of parallel tasks to a sequential task. These enable the control flow to transition to several parallel activities simultaneously and to ensure that all parallel tasks complete before proceeding to execute the next sequential task.

Our activity diagrams are code-based. Each activity represents an atomic WSL statement. Because the WSL code lines of a procedure form steps in the execution of this procedure and because individual WSL code lines form an atomic unit of execution, basing activities on individual WSL code lines is a logical basis for the nodes of an activity diagram. Conditions within WSL control constructs, such as WSL's if-then statements, form conditions within the guards that govern the flow of control to activities enclosed by the condition blocks of this WSL control construct (Yang & Ward, 2003).

One might question why TAGDUR chooses to generate activity diagrams from WSL code rather than simply allow the developers to view the WSL or generated C++ code of the transformed system. Activity diagrams were chosen for many reasons. UML is widely understood by many developers while WSL and, to a much lesser extent, C++ has less of a universal understanding. Furthermore, activity diagrams clearly represent the interaction among objects and the occurrences of events among activities; this representation would be much less apparent than if the developer were simply perusing the code of the transformed system.

Although our activity diagram is based on WSL code and individual WSL code lines are used to distinguish action states in the activity diagram, this lack of understanding is mitigated by attached comments which describe the WSL code line being modeled. For

Figure 15. An example of an activity diagram representation of the WSL code sample



example, a file I/O event is described in terms of its type (File I/O), subtype (Read), and destination, source, and index variables. The decision to base our activity diagram on WSL, rather than C++, was due to several reasons, including the fact that WSL is programming and platform independent. For example, a file I/O operation is represented through one type of WSL statement while representing the same operation in C++ may take several C++ statements depending on the type of file being accessed. Consequently, many implementation-specific details that would clutter an activity diagram based on C++ code are avoided if this same activity diagram is based on WSL code instead.

A small sample of WSL code is presented with a corresponding UML activity diagram based on this code.

WSL Code Sample:

```
X := Y + 1
If X > 4 Then
        Fetch D1, Y2, Z2
Else
        Call B.UpdateRec(X)
Fi
J := D1 + X
M := N + 2 /* potentially parallel operation */
K := 3          /* potentially parallel operation */
```

The following diagram models the following WSL code sample as action states in an activity diagram. Each action state is labelled by the WSL statement whose entry action

the state represents. Each action state is placed in the object swimlane whose object produces the action. For example, the invocation of Class B's UpdateRec method is represented as an action state in the Object B swimlane. Potential parallel operations, such as "M := N + 2" and "K:= 3," are modeled as parallel flows emanating from a fork synchronisation bar. An if-then-else WSL construct is modeled in the activity diagram as a branch, with mutually exclusive guard conditions, to two action states. Using these guard conditions, the control flow in the activity diagram is governed in a similar manner to the system that it represents. Potentially parallel executing WSL code lines are modeled as parallel control flows in the activity diagram (Millham & Yang, 2003b).

Statecharts

TAGDUR does not extract statecharts, similar to state diagrams, from source code but does derive activity diagrams, which describe a system's behaviour, from source code. Statecharts differ fundamentally from activity diagrams in that statecharts can only model elements of the system, such as classes or subsystems, whose behaviour can be formally described using a state machine while activity diagrams can model almost any process of the system. Although statechart modeling is a future feature of TAGDUR, statecharts may not be necessary to understand the behaviour of many legacy systems. While external events may be a critical part of legacy systems that are highly interactive and reactive, many legacy systems are batch-oriented. Because these systems are batch-oriented, the only event external to this type of system is the arrival of batch input that invokes the batch application. Consequently, in batch-oriented legacy systems, activity diagrams are sufficient to describe the behaviour of this system.

Use Case Diagrams

Use case diagrams are very necessary in order to clarify system requirements from the end-users. However, many other methods, such as textual representation or formal specifications methods such as Z, exist to depict the information contained within a use case diagram. Thus, while defining system requirements is a necessary part of the software development process, it is not strictly necessary to utilise use case diagrams to model this information.

Use case diagrams are very difficult to derive from source code during the reverse engineering process. While it is possible to extract system processes from activity diagrams and to properly label these processes using some type of artificial intelligence, it is very difficult to derive use cases without significant manual input from the users. For example, the purpose of a process may be derived through such means as natural language parsing and analysis of programmer comments associated with this process and of the names of associated procedures. If a procedure is named UPD-ACCT, the reverse engineering tool, through an analysis of this procedure name and concordance with programmer-defined abbreviation standards, may conclude that this procedure's purpose is to update a bank account. However, this natural language parsing and analysis process is often faulty. Furthermore, a reverse engineering tool cannot easily

know, without explicit user input, of actors external to the system. For example, the reverse engineering tool may detect places in the source code which accept user input but from this source code, the tool cannot easily determine what particular type of user might input this information.

Consequently, a reverse engineering tool cannot produce a relatively error-free use case diagram from source code. Instead, users must participate heavily in this process. These users must develop use cases from their own system knowledge and activity/class diagrams of this system. Because one of the goals of TAGDUR is to automate the derivation of UML diagrams from software architecture as much as possible, use case diagrams, because they require so much manual user intervention, are not yet an implemented feature of TAGDUR.

A Small Case Study

A small case study of a reengineering effort using TAGDUR is presented. The original legacy system is a COBOL, batch-oriented system of 1,500 source code lines (Hutty & Spence, 1997). This COBOL program, after its translation into the intermediate language WSL, is transformed into an object-oriented, event-driven system.

This transformed system is then analysed and various UML diagrams are derived from the code of this system. The following statistics are produced:

1. There are 22 classes identified. Of these 22 classes, 20 classes are derived from the application domain and 2 are predefined objects (System, File).
2. These classes have a total of 887 attributes and 40 methods. There are 627 associations, one class shares variables or methods of another class, between different classes.
3. There are 337 events identified. Of these events, the following types of events occur:

    - 1 external event (the invocation of the application upon arrival of input)
    - 19 error
    - 19 system interrupts
    - 220 file I/O operations
    - 113 method invocations

4. In the activity diagram derived from the source code, there are 1,275 action states identified and modeled. There are 41 conditions that govern the execution of these action states.

Conclusion

Creating a well-defined system model of a legacy system that is being reversed engineered is crucial in order to understand the structure and dynamics of the legacy system. Visualisation of this model, through a graphical modeling notation such as UML, has the advantage over textual notations in that graphical notations are better able to depict the complex relationships between model elements.

UML was chosen as the graphical modeling notation for several reasons, including UML's good tool support and its ability to support multiple perspectives of the same system.

However, many legacy systems are procedurally structured and driven. This structure and behaviour of this type of legacy system is difficult to model in UML because UML presupposes an object-oriented and, to some extent, an event-driven system. In order to represent a legacy system in UML, it is necessary to restructure the original system into an object-oriented system with the independent tasks and events of this system identified. This restructuring must occur using a set of proven restructuring processes in order to prevent restructuring errors from affecting the restructured system. Restructuring processes may be proven through proofs of correctness, functional equivalence of source and restructured systems, and many other methods. TAGDUR provides these proven restructuring processes of a legacy system in order to enable it to be modeled in UML and in order to achieve the advantages of an object-oriented system such as componentisation, lower maintenance costs, and easier program comprehension by developers through modularisation.

Deriving information from an analysis of the legacy system is a complex process. A theoretical framework to extract UML diagrams from the legacy system is provided. A partial implementation of this framework is available in the TAGDUR tool which extracts seven of the possible nine UML diagrams from the legacy systems. These seven diagrams represent the behavioural, static, dynamic, and architectural views of the system. However, TAGDUR does not extract statecharts or use case diagrams from the legacy systems. TAGDUR was originally designed to reverse engineer batch-oriented legacy systems from their only surviving information, their source code, and no user interaction during the reverse engineering process. Consequently, within this set of constraints, only a limited number of UML diagrams, which exclude use case and statechart diagrams, could be satisfactorily extracted. Statecharts, because they model event-response actions precisely in terms of states, are important in modeling and in understanding the behaviour of highly reactive systems. Activity diagrams, which are provided by TAGDUR, are often sufficient to model the behaviour of simpler batch-oriented legacy systems. Use case diagrams are crucial in modelling business process and in representing the system from an end-user viewpoint. However, use case extraction from the legacy systems requires a considerable amount of user intervention. Consequently, use case extraction has not been implemented in TAGDUR.

Enabling the developers to fully understand the structure, behaviour, and interaction with external devices of a system through reengineering and re-documentation through a series of selected UML diagrams allows the developers to recover lost documentation,

assist with maintenance such as existing error correction and prevention, facilitates software reuse by enabling the identification of reusable software components, enables the integration of disparate software systems, and enables the migration to another hardware/software platform (Klosch, 1996). Many tasks involving legacy systems, such as making changes to existing code or transporting it to a new platform, require a full understanding of the current system first in order to accomplish these tasks satisfactorily.

References

Alhir, S.S. (1998). *UML in a nutshell.* Sebastapol, CA: O'Reilly.

Ambler, S.W. (2002). UML class diagramming guidelines. Available online at *http://www.modelingstyle.info/useCaseDiagram.html*

Arango, G., Baxter, I., Freeman, P., & Pidgeon, C. (1986). TMM: software maintenance by transformation. *IEEE Software, 3*(3), 27-39.

Baecker, R. (1981). Sorting out sorting dynamic graphics project. University of Toronto: ACM SIGGRAPH '81 (distributed by Morgan Kaufmann, Los Altos, CA). *Software Visualization: Programming as a Multimedia Experience* (pp. 369-381). MIT Press.

Barstow, D. (1985). On convergence toward a database of program transformations. *ACM Transactions on Programming Languages and Systems, 7*(1), 1-9.

Ben-Menachem, M. & Marliss, G.S. (1997). *Software quality production practical, consistent software.* Boston: International Thomson Computer Press.

Bennett, S., McRobb, S., & Farmer, R. (2002). *Object-oriented systems analysis and design using UML* (2nd edition). Maidenhead: McGraw-Hill.

Biggerstaff, T.J. (1989). Design recovery for maintenance and reuse. *IEEE Computer, 22*(7).

Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The unified modeling language user guide.* Boston: Addison-Wesley-Longman.

Breuer, P.T. & Lano, K. (1991). Creating specification from code: Reverse-engineering techniques. *Journal of software maintenance: Research and practice.*

Chen, Y.F. & Ramanoorthy, C.V. (1986). The C information abstractor. *COMSASC 86*, 291-298.

Chifosky, E.J. & Cross, J.H.II (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software, 7*(1), 13-17.

Dorfman, M. & Thayer, R. (1997). *Software engineering.* Los Alamitos, CA: IEEE Computer Society Press.

D'Souza, D.F. & Wills, A.C. (1999). *Objects, components, and frameworks with UML.* Boston: Addison-Wesley.

Goldstein, H.H. & Neumann, J.V. (1949). *Planning and coding problems for an electronic computing instrument*. New York: McMillan.

Graham, I. (2000). *Requirements engineering and rapid development.* Boston: Addison-Wesley.

Graham, I. (2001). *Object-oriented methods principles & practice* (3rd edition). Boston: Addison-Wesley.

Hall, P.A.V. (1992). *Software reuse and reverse engineering in practice.* UK: Chapman & Hall.

Heywood, R. (2002). *UML use case diagrams: Tips and FAQ.* Available online at *http://www.andrew.cmu.edu/course/90-754/umlucdfaq.html#actors*

Howe, D. (2002). *FOLDOC: Free on-line dictionary of computing.* Available online at *http://foldoc.doc.ic.ac.uk/foldoc/index.html*

Hutty, R. & Spence, M. (1997). *Mastering COBOL programming.* London: Macmillan Press.

Irwin, W.N. & Churcher, N. (2002). XML in the visualisation pipeline. In D.D. Feng, J. Jin & P. Eades (Eds.), *Visualisation 2001, Vol. 11 of Conferences in Research and Practice in Information Technology*, Sydney, Australia, April, (pp. 59-68). Online at *http://citeseer.ist.psu.edu/article/irwin01xml.html*

Johnson, R.E. & Foote, B. (1988). Designing resusable classes. *Journal of Object-Oriented Programming. 1*(2), 22-35.

Johnson, W.L. (1986). *Intention-based diagnosis of novice programming errors.* Los Altos, CA: Morgan Kaufmann Publishers.

Klosch, R. (1996). Reverse engineering: Why and how to reverse engineer software. *Proceedings of CSS '96,* Los Angeles, (pp. 92-99).

Larman, C. (1998). *Applying UML and patterns — An introduction to object-oriented analysis and design.* Indianapolis, IN: Prentice-Hall.

Li, Y. & Yang, H. (2001). Simplicity: A key engineering concept for program understanding. *International Workshop on Program Comprehension (IWPC01).*

Lubara, M.D. (1991). Domain analysis and domain engineering in IdeA. In *Domain analysis and software systems modelling* (pp. 163-178). Los Alamitos, CA: IEEE Computer Society Press.

Millham, R. (2002). An investigation: Reengineering sequential procedure-driven software into object-oriented event-driven software through UML diagrams. *Proceedings of the International Computer Software and Applications Conference (COMPSAC),* Oxford.

Millham, R. &Yang, H. (2003). TAGDUR: a tool for producing UML diagrams through reengineering of the legacy systems. *Proceedings of the 7th IASTED International Conference on Software Engineering and Applications (SEA),* Marina del Rey, CA.

Millham, R., Pu, J., &Yang, H. (2004). TAGDUR: a tool for producing UML sequence, deployment, and component diagrams through reengineering of the legacy systems. *Proceedings of the 8th IASTED International Conference on Software Engineering and Applications (SEA)*, Innsbruck, Austria.

Millham, R., Yang, H., & Ward, M. (2003). Determining granularity of independent tasks for reengineering a legacy system into an OO system. *Proceedings of the International Computer Software and Applications Conference (COMPSAC),* Dallas.

Muller, P.A. (1997). *Instant UML.* Birmingham: Wrox Press.

Müller, H.A. (2004). *Understanding software systems using reverse engineering technologies research and practice.* Available online at *http://www.rigi.csc.uvic.ca/UVicRevTut/F6tools.html#Reengineering%20tool%20taxonomy*

Neighbors, J.M. (1984). The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering, SE-10*(5), 564-571.

Price, B., Small I., & Baecker, R. (1992). A taxonomy of software visualization. *Proceedings of the 25th Hawaii International Conference on System Sciences.*

Pu, J., Millham, R., & Yang, H. (2003). Acquiring domain knowledge in reverse engineering the legacy code into UML. *Proceedings of the 7th IASTED International Conference on Software Engineering and Applications (SEA),* Marina del Rey, CA.

Pu, J. & Yang, H. (2003). Modelling legacy code with UML class diagrams — Another reverse engineering attempt. *Proceedings of the Conference CACSUK2003,* Luton University, Luton, UK.

Rajlich, V. (1992). Redocumentation of software architecture. Position paper. In P.G. Selfridge et al. (Eds.), *Applying artificial intelligence to software problems* (pp. 7-14).

Reed, P. (1998). *The unified modelling language takes shape.* Colorado Springs, CO: Jackson-Reed.

Rubin, D.M.(1998). *Uses of use case.* Available online at *http://www.softstar-inc.com/index.htm*

Rugaber, S. & Clayton R. (1993). The representation problem in reverse engineering. *Proceedings of the 1993 Working Conference on Reverse Engineering.*

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorenson, W. (1991). *Object-oriented modelling and design.* Indianapolis, IN: Prentice-Hall.

Rumbaugh, J., Jacobson, I., & Booch, G. (1999). *The unified modeling language reference manual.* Boston: Addison-Wesley-Longman.

Srinivas, Y. (1991). *Pattern-matching: A sheaf-theoretic approach.* Unpublished Doctoral Thesis. Dept. of Information and Computer Science, University of California at Irvine.

Systa, T. (2000). *Static and dynamic reverse engineering techniques for Java software systems.* University of Tampere, Finland.

Systa, T. & Koskimies, K. (1997). Extracting state diagrams from the legacy systems. *ECOOP Workshops,* 272-273.

Van, S.L. (1992). *Workshop Notes – AI and Program Understanding.* AAAI.

Ward, M. (1992). *The syntax and semantics of the wide spectrum language.* Technical Report. Durham University, UK.

Ward, M., Calliss, F.W., & Munro, M. (1989). The maintainer's assistant. *Proceedings of Conference on Software Maintenance,* (pp. 307-315).

Webster, D.E. (1987). *Mapping the design representation terrain: A survey.* Technical Report. Micro-Electronics and Computer Technology Corporation. MCC STP-093-87.

Whitney, M. et al. (1995). Using an integrated toolset for program understanding. *Proceedings of the CASCON '95,* (pp. 262-274).

Wile, D.S. (1987). Local formalisms: Widening the spectrum of wide-spectrum languages. In *Program specification and transformation* (pp. 165-195). Burlington, MA: Elsevier Science Publishers.

Wills, L. (1993). Flexible control for program recognition. *Proceedings of the 1993 Working Conference on Reverse Engineering,* Baltimore, MD.

Yang, H. (1991). The supporting environment for a reverse engineering system — the maintainer's assistant. *IEEE Conference on Software Maintenance (ICSM 91).* Los Alamitos, CA: IEEE Computer Society Press.

Yang, H. & Ward, M. (2003). *Successful evolution of software systems*. Norwood, MA: Artech House.

Yang, H. et al. (1999). Acquisition of entity relationship models for maintenance — Dealing with data intensive programs in a transformation system. *Journal of Information Science and Engineering, 15*(2), 173-198.

Yang, H. et al. (2000). Abstraction: A key notion for reverse engineering in a system reengineering approach. *Journal of Software Maintenance: Research and Practice, 12*(5), 197-228.

Yang, H., Luker, P., & Chu, W. (1997). Measuring abstractness for reverse engineering in a re-engineering tool. *IEEE International Conference on Software maintenance,* 48-57.

Chapter XI

XML-Based Analysis of UML Models for Critical Systems Development

Jan Jürjens, TU München, Germany

Pasha Shabalin, TU München, Germany

Abstract

High-quality development of critical systems poses serious challenges. Formal methods have been proposed to address them, but their use in industry is not as widespread as originally hoped. This chapter proposes to use the Unified Modeling Language (UML), the de-facto industry standard specification language, as a notation together with a formally based tool-support for critical systems development. The authors extend the UML notation with new constructs for describing criticality requirements and relevant system properties, and introduce their formalization in the context of the UML executable semantics. Furthermore tool-support concepts for this approach are presented, which facilitate transfer of the methodology to industrial applications.

Introduction

Modern society relies on distributed IT-based infrastructures in many aspects including communication, finance, energy mining and distribution, and transportation. The disrup-

tion or incorrect functioning of these systems may threaten the economical or even physical well-being of people and organizations.

Complex distributed systems can have subtle flaws, which are not obvious and often cannot be detected by common testing procedures. Additionally, the distributed character of such infrastructures and the interconnection of modern information systems make remote and anonymous attacks on them possible. Examples indicating the potential scale of the problem include the following:

- The survey published in 2002 by Computer Security Institute in cooperation with FBI indicated that 90% of the interviewed organizations detected intrusion into their IT infrastructures within the last year. Only 44% of them were willing and able to quantify their losses. These 223 companies reported $455,848,000 in financial losses (Richardson, 2003).
- In 1997, a NASA hacker team broke into the U.S. Department of Defense and U.S. electric power grid system networks. They were able to provoke power outages and 911 emergency phone overloads in Washington, D.C. (Schneider, 1999).
- Spectacular examples for software failures in complex systems include problems with the Ariane 5 rockets: an independent inquiry board set up to investigate the explosive failure in 1997 reported that the flight control system failed because of errors in computer software design.

Obviously, the problem with critical systems was not left unnoticed, and many methodologies exist to improve their reliability. However, as we argue in the following section, we are still far from a satisfying solution. In this chapter, we would like to demonstrate how the Unified Modeling Language (UML) together with XML-based processing of UML models, offers a significant step toward a solution to the problem, in the context of model-based development of critical systems using UML.

Overview and Background

Traditionally, different methods exist for ensuring reliability of critical systems:

Break-And-Fix. This approach accepts that deployed systems may fail; whenever a problem is noticed and identified, the error is fixed. The Break-And-Fix approach is probably the most obvious one, however, it has many drawbacks. It is inherently disruptive — fixing the system often implies distributing patches, which disturbs users, annoys customers, and destroys their confidence. What is worse, the method is unsafe and insecure — we can never be sure that the new problem will not disturb critical functionality, or that it will not be spotted at first by a malicious person, who will try to compromise the system further.

Traditional formal methods, on the other hand, offer very good quality of the developed critical systems. There is much successful research in this direction. For security-

critical systems, this includes Millen, Clark, and Freedman (1987), Burrows, Abadi, and Needham (1989), Meadows, (1991), Lowe (1996), Abadi and Gordon (1997), and Paulson (1998). However, formal methods are rarely applied in practice because of the high costs arising from the necessary training for the developers of the system, and from the construction of the formal specification of the system.

UML, the de facto industry-standard in object-oriented modeling, together with XML-based processing of UML models, offers an unprecedented opportunity for high-quality critical systems development that is feasible in an industrial context.

- As the de facto standard in industrial modeling, a large number of developers are trained in UML, making less training necessary. Also, UML specifications of systems under development may already be available for analysis, which again saves time and cost.
- Compared to previous notations with a user community of comparable size, UML is relatively precisely defined, opening up the possibility for advanced tool-support to assist the development of safety-critical systems.
- After several years of evolution, an XML/XMI-based standard for UML model representation has evolved, enabling interchange and automated processing of the UML models.

XML-Based UML Analysis for Critical Systems Development

We will now explain how UML, together with an XML-based analysis of UML models, can be used as a basis for a formally based method for critical systems development. We will first analyze our requirements on the proposed method and demonstrate how the UML-based solution meets them. To keep the presentation concise and intelligible, we will restrict ourselves to security-critical systems. However, our approach is generic and can be applied to other criticality requirements such as safety and quality-of-service.

The UML-based formal methodology for development of security-critical system should meet the following requirements:

- Given a system model described with UML, it should *automatically* evaluate it for security-related vulnerabilities in design.
- The methodology should be available to developers *not specialized in security*, and still allow them to ensure the necessary security properties of the system under design.

- Security properties are often imprecisely defined or misunderstood. Formulating security properties of a system can often be a challenge by itself. Therefore we should enable the user to define easily and unambiguously both *security features* and *security requirements* of the system. The latter step is often considered as granted. However for many security properties it can be very difficult (Ryan, Schneider, & Goldsmith, 2001) and normally requires the developer to have special qualifications in cryptography.
- Costs of correcting flaws in a software system grow dramatically in the process of development; therefore, we would like to consider security from *early design phases*.
- Consider security on *different levels of abstraction* and *in system context*. Security of a complex distributed computer system can be violated on different levels. Even worse, security properties are generally not preserved by the composition (Jürjens, 2001). Therefore, blindly combining even proven security mechanisms may result in a faulty system. The method should detect these kinds of errors.
- Make use of the powerful pattern concept and encapsulate *established rules of prudent security engineering*.
- For certain security-critical software products, such as firewalls, the acceptance procedure is comparable to the development itself in laboriousness. Thus, we want to make *certification cost-effective*.

Now we will look closer at some of the requirements listed. It is obvious that today any software development methodology, which aims broad acceptance, needs to provide the end user with software tools supporting it. We were facing two challenges in this regard.

First, we need a uniform and standardized way of acquiring and processing UML models. Until recently, there were no standards on storing UML models, and different UML editing tools were producing files in proprietary format. The development and spreading of XML as a universal data representation language motivated the development of the XML Metadata Interchange (XMI) language, which is used, among other applications, for storing UML models into a file.

For developing critical systems using UML and XML-based analysis, one needs a precise semantics of the used notation. The UML is relatively precisely defined, however its semantics is given partially in prose, leaving room for ambiguities (UML Revision Task Force, 2001). We have refined the semantics by giving mathematically precise meaning of UML constructs, as shown in the next chapter.

Now we will take a close look at *UMLsec*, the UML-based language for secure-critical system development.

Creating and Using UMLsec

There is a set of requirements to meet before UML can be used for secure system development. The language must be extended with the necessary security-related

constructs. Correct application of the new language in the application domain must be described and enforced. The conflict between flexibility and non-ambiguity of the notation must be solved. Last but not least, tools for working with UMLsec models must be created. In more details, on the notational level we want the language to have the following properties.

Basic security requirements such as *secrecy* and *integrity* should be integrated into the language.

Different Threat scenarios should be considered automatically depending on the adversary strengths.

Common security concepts like *tamper-resistant hardware* should be readily available.

Common security mechanisms such as *access control* should be included in the language.

Cryptographic primitives (e.g., *(a)symmetric encryption*) should be defined and correctly handled. Data security properties of a computer system cannot be feasibly modeled at the abstraction level of the data values and cryptographic key values being processed. In fact, a possibility for precise modeling of a cryptographic algorithm at the data value level means finding a feasible way of breaking the algorithm, and therefore renders it useless. Thus, the modeling with UMLsec is normally done on the protocol level using the supplied Cryptographic primitives.

Physical security of the deployed system needs to be described by using the language.

Security management (e.g., *secure workflow*) should be addressed.

Domain-specific extensions (Java, smart cards, CORBA, ...) shall be considered during language design.

UMLsec: Extending UML

The UML specification (UML Revision Task Force, 2001) introduces *profiles* as a *lightweight* mechanism for extending the language (as opposed to *heavyweight* extensions through modifying the UML metamodel). A profile contains definitions for *stereotypes*, *tagged values,* and *constraints*. An important feature of a lightweight extension is that it should be "strictly additive to the standard UML semantics. This means that such extensions must not conflict with or contradict the standard semantics." (UML Revision Task Force, 2001). In particular, adhering to this requirement ensures that UMLsec can be used to extend any UML model without conflicting with existing tools or other UML extension, potentially UML profiles for other criticality requirements.

Stereotypes define new types of modeling elements extending the semantics of existing types in the UML metamodel. Their notation consists of the name of the stereotype written in double angle brackets << >>, attached to the extended model element. This model element is then interpreted according to the meaning ascribed to the stereotype.

Tagged values allow explicitly attaching a property to a model element. They are represented by a name=value pair in curly brackets associated with model elements. The value can be either a simple data type value, or a reference to another model element.

Constraints can be attached to a model element to refine its semantics. Attached to a stereotype, constraints must be observed by all model elements marked by that stereotype.

With UMLsec, stereotypes and tagged value are used to define data security requirements on model elements, and to define their security-relevant properties. The constraints, given using the formal semantics formulate rules, which must be met by the design to support the requested security properties. The most commonly used UMLsec stereotypes together with associated tags are listed in Figure 1.

To enable automatic reasoning about UMLsec model properties, formal semantics are introduced, which give mathematically precise meaning to the UML subset we are using, and to UMLsec constructs in its content. The following subset of UML diagrams is considered.

Class diagrams define the static class structure of the system: classes with attributes, operations, and signals and relationships between classes. On the instance level, the corresponding diagrams are called *Object diagrams*.

Statechart diagrams give the dynamic behavior of an individual object or component: events may cause a change in state or an execution of actions.

Sequence diagrams describe interaction between objects or system components via message exchange.

Figure 1. UMLsec stereotypes

Stereotype	Base Class	Tags	Description
Internet	link		Internet connection
encrypted	link		Encrypted connection
LAN	link, node		LAN connection
smart card	node		Smart card device
secure links	subsystem		Enforces secure communication links
secrecy	dependency		Assumes secrecy
integrity	dependency		Assumes integrity
high	dependency		Assumes high sensitivity, both secrecy and integrity
secure dependency	subsystem		Structural interaction data security
critical	object, subsystem	secrecy, integrity, high, fresh	Critical object
no down-flow	subsystem	secret	Prevents leak of information
fair exchange	subsystem	start, stop	After start eventually reaches stop
guarded access	subsystem		Access control using guard objects
guarded	object	guard	Guarded object

Activity diagrams specify the control flow between several components within the system, usually at a higher degree of abstraction than statechart diagrams and sequence diagrams. They can be used to put objects or components in the context of overall system behavior or to explain use cases in more detail.

Deployment diagrams describe the physical layer on which the system is to be implemented.

Subsystems (a certain kind of *packages*) integrate the information between the different kinds of diagrams and between different parts of the system specification.

Security Analysis of UMLsec Models

To apply formal verification methods for testing security properties of a distributed — which means open — system, it is necessary to "close" it by modeling all the possible interactions between the system and the outside world. This includes behavior of the potential adversary trying to break or compromise the system. For that reason, efficiency

Figure 2. Simulating adversary behavior

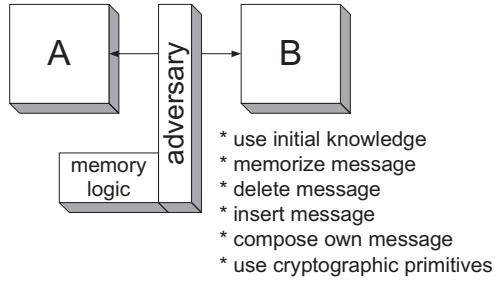


Figure 3. Threats from the default adversary

Stereotype	$Threats_{default}()$
Internet	***{delete, read, insert}***
encrypted	***{delete}***
LAN	∅
smart card	∅

Figure 4. Threats from the insider adversary

Stereotype	$Threats_{insider}()$
Internet	***{delete, read, insert}***
encrypted	***{delete}***
LAN	***{delete, read, insert}***
smart card	∅

and reliability of the automated verification of security properties depend on the correctness and completeness of the simulated adversary behavior. The issue is addressed in the literature, for example Lowe (1999) and Jacquemard, Rusinowitch, and Vigneron (2000). To create the adversary model we do not have to foresee any possibly attack scenario; we can define an intruder, which can do everything possible, as shown in Figure 2, and further restrict his behavior using additional information about the system.

The extent of possible intervention into the system functionality depends on the physical properties of the system, and on the adversary abilities. The UMLsec methodology provides possibility to define these parameters, addressing two issues. First, obviously impossible attack scenarios are not included in the analysis report. Second, the whole analysis process can be clustered into several steps, avoiding the state explosion problem (Clarke, Grumberg, & Peled, 1999).

We suggest that there can be adversaries with different capabilities regarding intervention with the system. Considering for example an online banking application, a simple user could read and modify information on the Internet, and access the client node. A malicious employee, however, could read and alter traffic in the internal LAN, and access internal servers. To define capabilities of an *adversary type A* against an object stereotyped with *s* we introduce a function $\mathbf{Threats_A(s)}$ returning a subset of *abstract threats* **{delete, read, insert, access}**. Examples in Figures 3 and 4 define the function for two adversary types; new adversary types can be freely formulated.

Basing on these model-wide definitions, we define how the adversary can interact with each model element. For a link *l* in deployment diagram contained in the subsystem *S* we define the set $\mathbf{threats^S_A(l)}$ of concrete threats to be the smallest set satisfying the following conditions. If each node *n*, that *l* is contained in (note that nodes and subsystems may be nested one in another), carries a stereotype s_n with $\mathbf{access \in Threats_A(s_n)}$, then:

- If *l* carries a stereotype *s* with $\mathbf{delete \in Threats_A(s)}$, then $\mathbf{delete \in threats^S_A(l)}$.
- If *l* carries a stereotype *s* with $\mathbf{insert \in Threats_A(s)}$, then $\mathbf{insert \in threats^S_A(l)}$.
- If *l* carries a stereotype *s* with $\mathbf{read \in Threats_A(s)}$, then $\mathbf{read \in threats^S_A(l)}$.
- If *l* is connected to a node that carries a stereotype *s* with $\mathbf{access \in Threats_A(s)}$, then $\mathbf{\{delete, read, insert\} \subseteq threats^S_A(l)}$.

The idea is that $\mathbf{threats^S_A(x)}$ specifies the threat scenario against a component or link *x* in the subsystem *S* that is associated with an adversary type *A*. On the one hand, the threat scenario determines which data the adversary can obtain by accessing components; on the other hand, it determines which actions the adversary is permitted by the threat scenario to apply to the concerned links. **delete** means that the adversary may delete the messages on the corresponding link, **read** allows him to read the messages on the link, and **insert** allows him to insert messages in the link. The new messages can be created by applying cryptographic primitives to the previously read data and initial knowledge of the adversary.

To investigate security of the system with respect to the chosen *adversary type* we build an executable specification of the system combined with the adversary model, and verify its properties.

This requires mathematically precise definitions of the UMLsec models, including their dynamic behavior. This is defined in Jürjens (2004) using *UML Machines*, which are inspired by the Abstract State Machines (ASM) (Gurevich, 1995; Börger & Stärk, 2003). A UML Machine is a transition system the states of which are algebraic structures, with built-in communication mechanisms; it defines behavior of a system component. A UML Machine is defined by the *initial state*, *transitions rules*, which is applied iteratively and defines how the machine state changes in time, and two multi-set buffers (*output queue* and *input queue*).

A UML Machine communicates with other system parts by adding messages to its output queue and retrieving messages from its input queue. All the messages in the system compose the set **Events**. To build executable UML specification in a modular way, by combining a set of UML Machines together with communication links connecting them, one can use the notion of a *UML Machine System* (UMS) also defined in Jürjens (2004). The intuition is that a UMS models a computer system that is divided into components that may communicate by sending messages through communication links and whose execution is scheduled by a specified scheduler.

The definition of the UML semantics has to be omitted here and can be found in Jürjens (2004).

Example Application

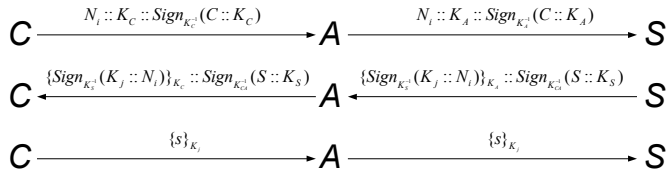
We demonstrate usability of the UMLsec methodology on a variant of the Internet security protocol TLS (the successor of SSL) as proposed in Apostolopoulos, Peris, & Saha (1999). The example in Figure 5 shows the UMLsec specification.

The goal is to let a client C send a secret m over an untrusted communication link to a server S in a way that provides confidentiality and server authentication, by using a symmetric key K_{CS} to be exchanged.

We write $\{E\}_K$ for the encryption of the expression E under the key K and $Dec_K(E)$ for the decryption of E with K. The protocol uses both RSA encryption and signing. Thus. we assume the equations $Dec_K^{-1}(\{E\}_K) = E$ and $\{Dec_K^{-1}(E)\}_K = E$ to hold (where K^{-1} is the private key belonging to the public key K).

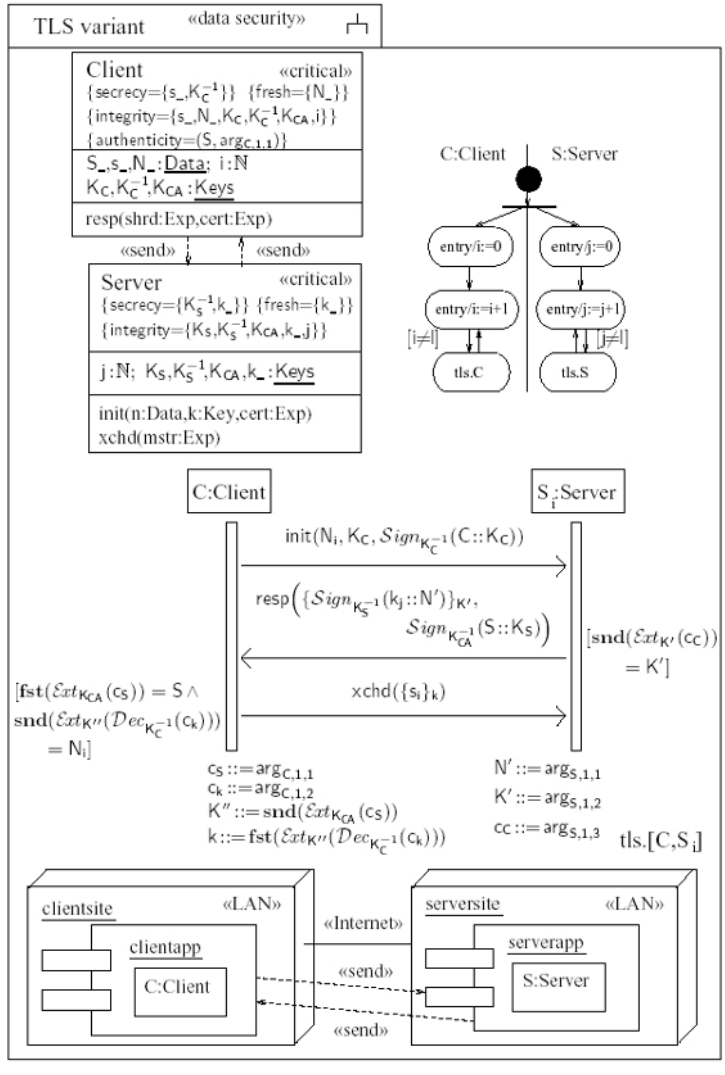
The protocol assumes that there is a secure (with respect to integrity) way for C to obtain the public key K_{CA} of the certification authority, and for S to obtain a certificate $Dec_{Kca}^{-1}(S :: K_S)$ signed by the certification authority that contains its name and public key. The adversary also may have access to K_{CA}, $Dec_{Kca}^{-1}(S :: K_S)$ and $Dec_{Kca}^{-1}(Z :: K_Z)$ for an arbitrary $Z \in \textbf{\textit{Data}} \setminus \{S\}$. The complete specification can be found in Jürjens (2004).

One can now demonstrate that this proposed variant of TLS contains a security flaw. The message flow diagram corresponding to the man-in-the-middle attack is the following.



Here the adversary has access to the communication link between client and server since it is supposed to be an Internet link. More details are given in Jürjens, (2004) as well as a correction which is proved secure using the formal semantics.

Figure 5. Variant of the TLS handshake protocol



Tools for Advanced XML-Based Processing of UML models

During the last couple of years UML is getting a completely new role in the software development. Traditionally since its appearance UML encapsulated two aspects:

- **Standardized notation** helped to capture, store and exchange knowledge about the system under design.
- **Semantics**, although semi-formal, ensured that different developers understood the common meaning of UML diagrams.

However, the information was meaningful on the graphical (diagram) level only. Different UML tools implemented proprietary UML storage format which made exchange and reuse of the models impossible. Having chosen a UML tool, the developer was tied to using it through the whole project. Applying emerging technologies to the UML modeling on the industrial level was virtually impossible. To suggest any custom UML processing, one would have to develop a complete UML editor and persuade the auditorium to use it.

Development of the XML as universal data storage format changed this situation dramatically. In the year 2000, the Object Management Group (OMG) issued the first specification for the XML Metadata Interchange (XMI) language, which — among other applications — became a standard for serializing UML models into a file.

The XMI language is compliant with Meta Object Facility (MOF), which is a framework for specifying meta-information (also called metamodels). Initially, it was developed to define CORBA-based services for managing meta-information. Currently, its applications include definition of modeling languages such as UML and Common Warehouse Model (CWM). The framework operates on a four-level data abstraction model, shown in Figure 6.

We consider the abstraction levels from bottom up. The lowest level **M0** deals with the data instances, for example *Mr. Smith, 35 years old, lives in New York*. The level **M1** describes data models, in software development this corresponds to the UML model of the application. An example for this layer is a *Person* with attributes *Name, Age, Address*. The next abstraction level **M2** is the modeling language itself. Different modeling

Figure 6. MOF framework

M3	Meta-Metamodel	**MetaClass, MetaAssociation** – MOF Model
M2	Metamodel	**Class, Attribute, Dependency** UML (as a language), CWM
M1	Model	**Person, City, Book** – UML Model
M0	Data	**Bob Marley, Bonn** – Running program

languages exist for different application domains, and the last abstraction level **M3** is the common environment for defining these modeling languages, standardized by the MOF. It operates with three elements:

MOF Object defines object types for the target model. It includes a *name*; a set of *attributes*, both predefined and custom; a set of *operations*; a set of *association references*; a set of *supertypes* it inherits; and some other information. The MOF object is a *container* for its component features (i.e., any *attributes*, *operations,* and *association references*). It also may contain MOF definitions of *data types* and *exceptions*.

MOF Association defines a link between two MOF objects. The MOF links are always binary and directed. A link is a *container* for two *association ends*, each representing one object the link is connected to.

MOF Package groups related MOF elements for reuse and modularization. It is defined by a *name*; a list of *imports* which defines a set of other MOF Packages whose components may be re-used by components defined within the Package; a list of *supertypes* which defines a set of other MOF Packages whose components form a part of the Package; and a set of contained elements including other Objects, Associations, and Packages.

The MOF also defines the following secondary elements:

Data Types can be used to define *constructed* and *reference* data types.

Constants define compile-time constant expressions.

Exceptions can be raised by Object operations.

Constraints can be attached to other MOF Elements. Constraint semantics and verification are not part of the MOF specification, and therefore they can be defined with any language.

The MOF is related to two other standards.

XML Metadata Interchange (XMI) is a mapping from MOF to XML. It can be used to automatically produce an XML interchange format for any language described with MOF. For example, to produce a standardized UML interchange format, we need to define the UML language using MOF and use the XMI mapping rules to derive DTDs and XML Schemas for UML serialization. MOF itself is defined using MOF itself, and therefore XMI can be applied not only for metamodel instances, but for metamodels themselves (as they also are instances of a metamodel, which is MOF).

Java Metadata Interface (JMI) standard defines MOF-to-Java mapping (similarly to the MOF-to-XML mapping provided by XMI). It is used to derive Java interfaces tailored for accessing instances of a particular metamodel. As MOF itself is MOF-compliant, it can be used to access metamodels, too. The standard also defines a

set of *reflective* interfaces that can be used similarly to the metamodel-specific API without prior knowledge of the metamodel.

After the standards were introduced, major producers of UML editors eventually have picked it up, and currently support model interchange in the XMI format. Together, with the wide support for the XML language in the industry including broad range of libraries, editors, and accompanying technologies, this enables development of the lightweight UML processing tools, tailored to carry one particular task.

The whole story is applicable to the formalized critical system development with UML. To facilitate acceptance of the formalized UML-based software development, automated processing of UML models was highly required. Prototype tools supporting this functionality have been developed at the TU Munich; some results of these projects are presented. Especially we hope that the publicly available Web-based interface will provide a simple and accessible entry into the methodology.

XML-Based Data-Binding with MDR

Technically the central question was how to work with UML/XMI files. Three possible approaches exist:

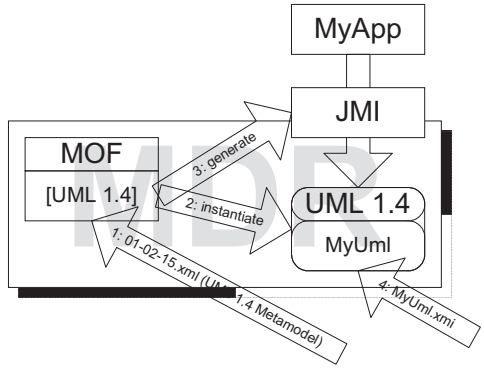
- XML parsing and transformation languages coupled with the XML standard (XPath, XSLT)
- Any high-level language with appropriate libraries (Java, C++, Perl)
- Data binding

The first two methods, although more flexible, require more development effort. However, for UML processing, we are concerned about the data contained in documents rather than about the document itself and its structure. For this purpose, data binding offers a much simpler approach to working with XML data.

Several libraries support data binding for XML. It was important to use one with appropriate data abstraction level. For example, the widely used Castor library would leave the developer with a very abstract representation of the UML model, on the level of MOF constructs. However existing data binding libraries provide representation of a UML / XMI file on the abstraction level of a UML model. This allows the developer to operate directly with UML concepts (e.g., classes, statecharts, stereotypes, etc.). We use the MDR (MetaData Repository) library, which is part of the Netbeans project, also used by the freely available UML modeling tool Poseidon 1.6 Community Edition. Another such library is the Novosoft NSUML project.

The MDR library implements a MOF repository with support for XMI and JMI standards. Figure 7 illustrates how the repository is used for working with UML models.

Figure 7. Using the MDR library



The XMI description of the modeling language is used to customize the MDR for working with a particular model type, in this care, UML (Step 1). The XMI description of the UML 1.4 is published by the Object Management Group (OMG). A storage customized for the given model type is created (Step 2). Additionally, based on the XMI specification of the modeling language, the MDR library creates the JMI (Java Metadata Interface) implementation for accessing the model (Step 3). This allows the application to manipulate the model directly on the conceptual level of UML. The UML model is loaded into the repository (Step 4). Now, it can be accessed through the supplied JMI interfaces from a Java application. The model can be read, modified, and later saved into an XMI file again.

Because of the additional abstraction level implemented by the MDR library, using it in the UML suite should facilitate upgrading to upcoming UML versions and promises the highest available standard compatibility.

XML-Based UML Tool Suite

Further, we present the architecture of the UML tool suite developed at the TU Munich. Its architecture and basic functionality are illustrated in Figure 8.

The developer creates a model and stores it in the UML 1.4 / XMI 1.2 file format. The file is imported by the UML into the internal MDR repository. Other components of the UML suite access the model through the JMI interfaces, generated by the MDR library. The Static Checker parses the model, verifies its static features, and delivers the results to the Error Analyzer. The Dynamic Checker translates the relevant fragments of the UML model into an input language of an external tool (Model Checker, Automatic Theorem Prover, Prolog). The external tool is spawned by the UML suite as an external process; its results are delivered back to the Error Analyzer. The Error Analyzer uses the information received from both Static Checker and Dynamic Checker to produce a Text Report for the developer describing found problems, and a Modified UML Model, where the found errors are visualized and/or corrected.

The idea behind the tools suite is to provide a common programming framework for the developers of different verification modules (*tools*). Thus, a tool developer should

concentrate on the verification logic and not on the handling Input/Output. Different tools, implementing verification logic modules (*Static Checkers* or *Dynamic Checkers* in Figure 8) can be independently developed and integrated. Currently, Static Checkers exist for most static UMLsec properties, and Dynamic Checkers for some dynamic properties.

The tool implementation follows the following simple concepts:

- It is given a default UML model it operates on. It may load further models if necessary.
- The tool exposes a set of commands, which it can execute.
- Every single command is not interactive. It receives parameters, executes, and delivers feedback.
- The tool can have its internal state which is preserved between commands.

The tool architecture presented in Figure 9 allows development of the verification logic independently of the input and output media with minimum effort. Each tool is required to implement the **ITextMode** interface, which exposes tool functionality in text mode, with string array as input and text as output. The framework provides default wrappers for

Figure 8. UML tools suite

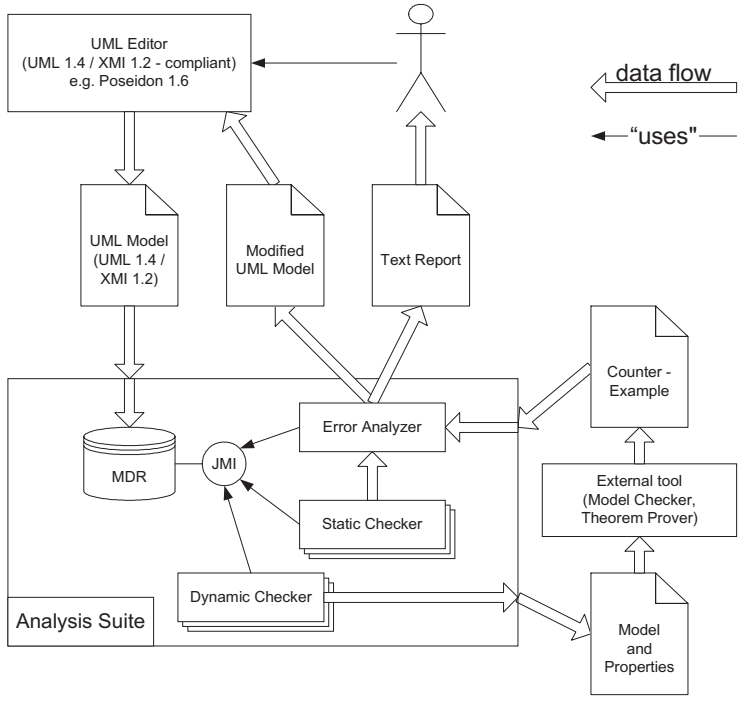
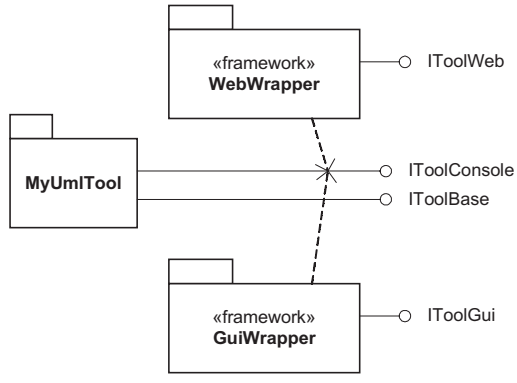


Figure 9. Tool interfaces



Graphical User Interface (GUI) **GuiWrapper** and Web mode **WebWrapper**. These wrappers enable using the tool without modifications in the GUI application (part of the framework) or through a Web interface by rendering the output text on the respective media. However, each tool may itself implement the **IGuiMode** and/or **IWebMode** to fully exploit the functionality of the corresponding media, for example, to fully use GUI mode capabilities to display graphical information.

Following these principles, the framework was initially developed for use in the text mode, and later a Web front-end was added to make the verification functionality available over the Internet. This extension did not require changes to the verification modules.

Future Perspectives

The proposed method of XML-based Analysis of UML Models has been successfully tried out in several industrial projects. The next goal in the development of the methodology is to extend the UML tool suite to provide complete support for security properties verification of UMLsec models.

However, the presented ideas are not exhaustive, at least the following future development directions seem interesting.

- The application of the methodology for further criticality requirements in the computer systems development, such as safety and real-time.
- The automatisation of other aspects of UML-based development. For example, Houmb and Jürjens (2003) combine UMLsec with model-based risk assessment. Broad practical acceptance of these approaches can be greatly facilitated by sufficient formalization and automated tool-support.

Conclusion

The development and spreading of the UML language within the last years, especially its standardization and introduction of supporting technologies, is changing its role in the software development from a notational aid to a powerful framework with support for automation of many development tasks.

We believe that the suggested approach to critical system development using UML and XML-based processing of UML models will find widespread acceptance in the modern software development industry for the following reasons:

- It is based on UML, which is the de-facto standard in software development, which facilitates acceptance in industrial software development teams.
- The application of the methodology requires no special training in security (in case of UMLsec) or the other criticality domains.
- The suggested formal semantics for a simplified fragment of UML lays a foundation for advanced XML-based tool-support for the methodology, making automatic verification of the criticality features possible.

References

Abadi, M. & Gordon, A.D. (1997). A calculus for cryptographic protocols: The Spi calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security* (pp. 36-47). New York: ACM Press.

Apostolopoulos, V., Peris, V., & Saha, D. (1999). Transport layer security: how much does it really cost? *Conference on Computer Communications (IEEE Infocom)*, New York.

Börger, E. & Stärk, R. (2003). *Abstract state machines*. Germany: Springer.

Burrows, M., Abadi, M., & Needham, R. (1989). A logic of authentication. *Proceedings of the Royal Society of London A*, *426*, (pp. 233-271). A preliminary version appeared as Digital Equipment Corporation Systems Research Center report No. 39, February.

Castor (2004). *Castor library.* Available online at *http://castor.exolab.org*

Clarke, E.M., Jr., Grumberg, O., & Peled, D.A. (1999). *Model checking*. Cambridge, MA: The MIT Press.

Gentleware (2004). *Gentleware corporation.* Available at *http://www.gentleware.com*

Gurevich, Y. (1995). Evolving algebras 1993: Lipari guide. In E. Börger (Ed.), *Specification and Validation Methods* (pp. 9-36). New York: Oxford University Press.

Houmb, S. &Jürjens, J. (2003). Developing secure networked web-based systems using model-based risk assessment and UMLsec. *10th Asia-Pacific Software Engineering Conference (APSEC 2003)*, Chiangmai (Thailand). IEEE Computer Society.

Jacquemard, F., Rusinowitch, M, & Vigneron, L. (2000). *Compiling and verifying security protocols.* Technical Report, LORIA Universite, France. Germany.

Jürjens, J. (2001). Composability of secrecy. In V. Gorodetski, V. Skormin, & Popyack, L. (Eds.), *International workshop on mathematical methods,models and architectures for computer networks security (MMM-ACNS 2001), Volume 2052 of LNCS* (pp. 28-38). Springer.

Jürjens, J. (2004). *Secure systems development with UML*. Springer.

Lowe, G. (1996). Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and algorithms for the construction and analysis of systems, Volume 1055 of LNCS,* (pp. 147-166). Springer.

Lowe, G. (1999). Towards a completeness result for model checking of security protocols. *Journal of Computer Security*, *7*(2,3).

Meadows, C. (1991). A system for the specification and analysis of key management protocols. In *IEEE Symposium on Security and Privacy* (pp.182-195).

Millen, J., Clark, S., & Freedman, S. (1987). The interrogator: protocol security analysis. *IEEE Transactions on Software Engineering, SE-13*(2), 274-288.

Netbeans (2004). *Netbeans project. Open source.* Available online at *http://mdr.netbeans.org/*

Novosoft (2004). *Novosoft NSUML project.* Available online at *http://nsuml.sourceforge.net/*

Object Management Group (2002). MOF 1.4 specification. Available at *http://www.omg.org/technology/documents/formal/mof.htm*

Object Management Group (2002a). *OMG XML metadata interchange (XMI) specification.* Available online at *http://www.omg.org/cgi-bin/doc?formal/2002-01-01*

OMG (2003). Object Management Group. Available at *http://www.omg.org/*

Paulson, L. C. (1998). The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, *6*(1-2), 85-128.

Richardson, R. (2003). *2003 CSI/FBI computer crime and security survey.* Technical report, Computer Security Institute, San Francisco. Available online at *http://www.gocsi.com/forms/fbi/pdf.html*

Ryan, P., Schneider, S., Goldsmith, M., Lowe, G., & Roscoe, B. (2001). *The modeling and analysis of security protocols: The CSP approach*. Boston: Addison-Wesley.

Schneider, F., editor (1999). *Trust in cyberspace*. National Academy Press. Available at *http://www.nap.edu/readingroom/books/trust*

UML Revision Task Force (2001). *OMG UML Specification v. 1.4.* OMG Document ad/01-09-67. Available at *http://www.omg.org/uml*

Chapter XII

Augmenting UML to Support the Design and Evolution of User Interfaces

Chris Scogings, Massey University, New Zealand

Chris Phillips, Massey University, New Zealand

Abstract

The primary focus in UML has been on support for the design and implementation of the software comprising the underlying system. Very little support is provided for the design or evolution of the user interface. This chapter commences with a brief review of UML and its support for user interface modeling. Lean Cuisine+, a notation capable of modeling both dialogue structure and high-level user tasks, is described. It is shown through a case study that Lean Cuisine+ can be used to augment UML and provide the user interface support that is currently lacking.

Introduction

Considerable effort has been devoted to the development of models and tools to support the analysis and design of software systems, culminating in the Unified Modeling Language (UML) (Rumbaugh, Jacobson, & Booch, 1999). The primary focus has been on support for the design and implementation of the software comprising the underlying

system, referred to in (Collins, 1995) as the *internal system*. Very little support is provided for the design of the *external system* or user interface.

The user interface is a vital part of any software application. To the user, the interface *is* the system (Collins, 1995). It is important, therefore, to devote time and effort to the user interface when building and upgrading software. Indeed, research indicates that applications are often upgraded precisely because a new user interface is desired (Church & te Braake, 2002). The user interface may be redesigned to improve its usability, to add new functionality, or to take advantage of new technologies and visualisation techniques – for example, a system may be changed from a forms-based interface to a graphical user interface (GUI), or a help system may be added.

A user interface can be described at three levels:

- the visible interface (the 'look and feel');
- the underlying structure and behaviour, known as the *dialogue*, which describes the response of the system to user and system generated events;
- the high-level user *tasks* (sequences of actions) that must be supported by the interface.

These three views of the user interface need to be identified and separated so that during software evolution the visible interface can be changed independently of dialogue structure and user tasks — although these also can be modified or extended if required. The dialogue and task information is preserved through the use of appropriate *conceptual models*.

This chapter commences with a brief review of UML and its support for user interface modeling. Lean Cuisine+, a notation capable of modeling both dialogue structure and high-level user tasks, is then introduced. It is shown through a case study that Lean Cuisine+ can be used to augment UML by providing the user interface support that is currently lacking. The work is placed in context.

UML and the User Interface

Prior to the introduction of UML, it had already been established that there was a lack of integration between the software engineering and HCI communities (Jacquot & Quesnot, 1997). Janssen, Weisbecker and Ziegler (1993) make the point that user interface tools cannot make use of the models developed with general software engineering methods and tools. Kemp and Phillips (1998) show that support for user interface design is weak within object-oriented software engineering methods.

As UML is primarily a collection of previously defined notations, no significant change has occurred. The approach taken in Eriksson and Penker (1998) is typical in that they state that the design of the user interface should be carried out separately but in parallel

to the design work carried out with UML. Quatrani (2000) mentions that prototyping the user interface is useful and adds that UML can be used to model something like a window but could not successfully be used for modeling each of its dialogue boxes or buttons.

Some of the UML notations are being used for user interface modeling. These include statecharts, use cases, and activity diagrams. However, Kovacevic (1998) states that UML activity diagrams and interaction diagrams (collaboration and sequence diagrams) are not enough to perform the task modeling required for user interface design. This is supported by Rourke (2002) which states that UML diagrams are not sufficient for task modeling and need to be linked to separate task models.

The UML software design process should incorporate user interface design as an important component. It is therefore desirable to include user interface modeling within the UML framework.

Augmenting UML to Support User Interface Modeling

In a discussion of this nature, it is necessary to clarify the difference between *strategies* and *notations*. UML is a collection of specific notations (a language), and this chapter discusses whether these notations are useful and efficient for modeling a user interface. There also exist strategies, or patterns, which provide guidelines for user interface design and layout but (deliberately) do not provide specific notations. Such patterns can be used with UML notations but could equally be used with any other suitable notations.

One such strategy is the Model-View-Controller or "MVC Architecture." It originally appeared in the 1980s and has undergone a number of changes over the years. It is still relevant today as a strategy but does not provide any notations. An interesting debate about the history and usefulness of MVC can be found at (MVC Design, 2003).

Researchers have suggested using UML notations to model the user interface, for example, OVID (Roberts, Berry, & Isensee, 1997) and ASP (Conallen, 2002) but such methods perpetuate the problem of using notations that are not intended for user interface modeling and they also use state diagrams to model dialogue when they are known to be inadequate (Phillips, 1993).

Naked Objects (Pawson & Matthews, 2002) supplies both a principle and a method. The principle is that all objects should be accessible to the user in the simplest possible way. However, the technique has several disadvantages:

- It is applicable only to a limited set of "database-type" applications and would be unsuitable for any graphics-based application such as a simulation. The authors themselves stress that it was developed for business applications;

- It is only useful for users who are familiar with the business concepts that are being modeled. For example, the user interface of a typical Automatic Teller Machine (ATM) could not be replaced by a naked object interface;
- The method is very limiting as a fixed interface is always used.

In order to successfully model a user interface within the UML environment, it is therefore necessary to provide a new notation that integrates well with UML, while augmenting its capabilities. One such approach is UMLi (Pinheiro da Silva & Paton, 2003) in which UML activity diagrams are extended to model tasks. However, UMLi still uses two separate diagrams — one to model the dialog and another to model tasks. This chapter introduces Lean Cuisine+ which is capable of modeling tasks within the dialogue structure of the user interface, and which combines naturally with UML.

Task and Dialogue Modeling

Task models focus either on task decomposition or task flow. Their primary purpose is to define the activities of the user in relation to the system, as a means of uncovering the functional requirements to be supported by the system. Task modeling is not new, and methods such as hierarchical task analysis (HTA) have been used for many years. Task modeling is also used in a wider context than that of modeling the user interface. For example, it appears in UML via *use cases* that have been so successful that they have been integrated into virtually every major approach to object-oriented analysis and design (Constantine & Lockwood, 1999).

Dialogue models have been developed as a means of capturing information about the behaviour of the user interface at an abstract level, separate from that of the visible interface. Research has found that it is necessary to focus on the structure of the dialogue to avoid producing a user interface that looks good but is functionally useless (Jacquot & Quesnot, 1997). Dialogue modeling has been used as the basis for various attempts at the automatic generation of user interfaces and tends to use well-known notations such as state transition diagrams (STDs) and Petri nets. It also can be carried out within UML by using statecharts. However, dialogue modeling using these formal notations is complicated and liable to error (Phillips, 1993).

A major problem with the modeling techniques and notations mentioned is that task and dialogue modeling are performed largely in isolation. Task models, such as use cases, provide no indication of user interface structure. Similarly, a criticism of existing dialogue notations is that the linkage to tasks is often obscure (Brooks, 1991). They define the structure and behaviour of the dialogue but they do not always readily reveal the interconnections between dialogue components during the execution of higher level tasks. That is, they do not represent tasks within the context of the dialogue structure. Ideally, task and dialogue models should be linked within a common notation.

Lean Cuisine+

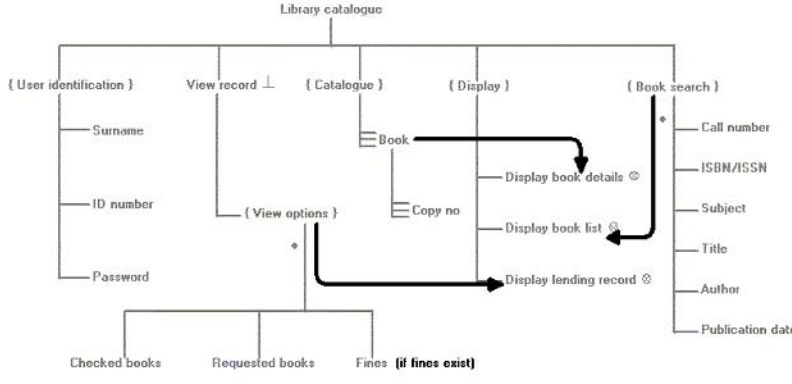
Lean Cuisine+ (Phillips, 1995) is a graphical notation for modeling the behaviour of event-based direct manipulation interfaces. The Lean Cuisine+ notation has several basic strengths in that it has been specifically developed as an interface design tool, it incorporates the structure of the interface dialogue and it fits easily into an object-oriented approach. A major advantage of the notation is that task sequences can be represented in the context of the structure of the interface dialogue and in this way, Lean Cuisine+ combines both dialogue modeling and task modeling. Lean Cuisine+ has been derived from Lean Cuisine (Apperley & Spence, 1989) and is an example of a *dialogue modeling language*.

In a Lean Cuisine+ specification, the interface is represented primarily as a dialogue tree. The behaviour of the interface is expressed in terms of the constraints and dependencies which exist between the nodes of the tree (the menemes). A meneme, which can represent an object or action available in the user interface, is defined as having just two possible states, "selected" and "not selected." Menemes can be grouped into subsets that are either mutually exclusive (1-from-N) or mutually compatible (M-from-N). Mutually exclusive options are shown diagrammatically by a horizontal fork, and mutually compatible options by a vertical fork.

An example showing the notation in use appears in Figure 1. This Lean Cuisine+ diagram models the user interface for a typical library catalogue system. Users are provided with several options, such as View record or Book search. Certain menemes act as headers of *subdialogues* – for example, User identification is the header of the subdialogue containing Surname, ID number, and Password. A *virtual meneme* is one where the name appears in braces. This indicates that the meneme is simply a header and is not available for selection. For example, {Display} is a virtual meneme but View record is not.

Selection triggers can be added to the diagram. These indicate where the selection of a particular meneme will generate the automatic selection (or deselection) of other meneme(s). In Figure 1, the trigger from Book to Display book details indicates that the

Figure 1. The Lean Cuisine+ tree for a typical library catalogue



selection of Book by the user will trigger the selection of Display book details by the system. The trigger from {View options} to Display lending record indicates that the selection of one or more of the menemes in the {View options} subdialogue will trigger the selection of Display lending record by the system. Note that {View options} itself can never be selected as it is a virtual meneme. Similarly, the third trigger indicates that the selection of one of the menemes in the {Book search} subdialogue will trigger the selection of Display book list.

Conditions can also be added to the diagram. For example, in Figure 1, a condition indicates that the meneme Fines may only be selected if the user has actually received one or more fines. Meneme designators also are added to the diagram at this stage. Most menemes can be selected or deselected by the user, and the designators indicate the exceptions to this rule. A *monostable* meneme (⊥) can be selected by the user, but then reverts to an unselected state on completion of the operation it represents. A *passive* meneme (⊗) cannot be selected or deselected by the user but can be selected by the system. A dot next to a subdialogue (see {View options}) indicates an *unassigned default* choice that takes on the value of the last user selection from the subdialogue.

Figure 2 shows the Lean Cuisine+ diagram from Figure 1 with a task sequence superimposed on it. Note that selection triggers have been hidden in this view. The task sequence displayed shows the selections needed for a library user to view their lending record. A solid arrow in the task indicates a user action and a dashed arrow indicates a system action. The task sequence is a representation of a UML *use case*.

Lean Cuisine+ is naturally more inclined toward object-oriented design than STDs, statecharts, or Petri nets. When tasks are modeled in conjunction with dialogue, Lean Cuisine+ presents the objects available for selection during task execution, whereas the other representations focus on the transitions between states in the system and thus emphasise task flow.

Lean Cuisine+ is a powerful tool for linking dialogue and task modeling. As an example, the Lean Cuisine+ task sequence in Figure 2 can be compared with the UML activity diagram representing the same task in Figure 3. This illustration clearly demonstrates the difference between the isolated task in the activity diagram and the task shown in the context of the dialogue in the Lean Cuisine+ representation.

Figure 2. A Lean Cuisine+ diagram showing a task sequence

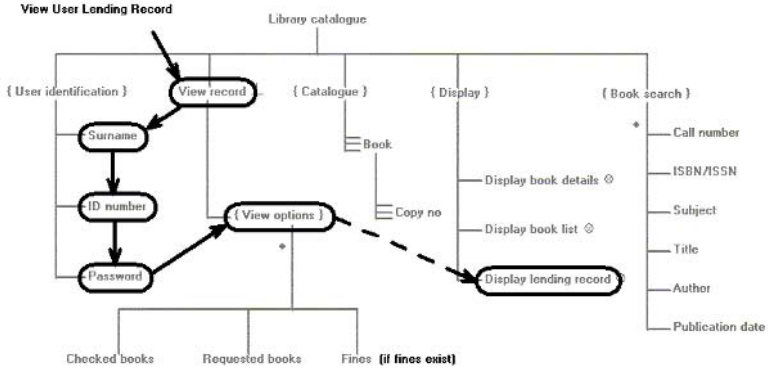
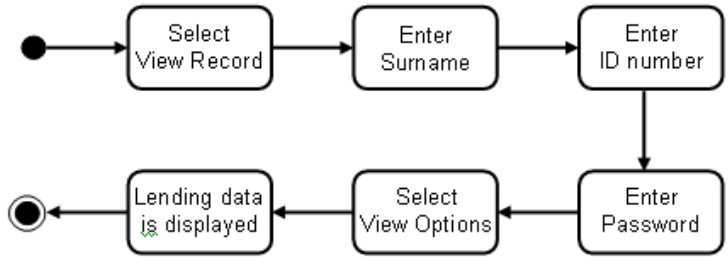


Figure 3. UML activity diagram for the task represented in Figure 2



The inclusion of task modeling provides several benefits: the task is represented at a level useful for user interface design; the task is shown within the context of the interface dialogue; system actions are clearly distinguished from user actions; and it can be used as an analysis tool, for example, to minimise key strokes in a task sequence.

Placing Lean Cuisine+ within the UML Context

The UML process for modeling software applications (known as the Unified Process) and the process for modeling user interfaces with Lean Cuisine+ have significant overlap. In particular, both processes begin with UML *use cases* and *class diagrams*. Thus, if a system is undergoing significant changes, it is possible to provide new UML use cases and class diagrams for the application and then derive Lean Cuisine+ models from these initial diagrams. A user interface can then be constructed from the Lean Cuisine+ models. A detailed description of the process for deriving a Lean Cuisine+ model from UML use cases and class diagrams is provided in Scogings and Phillips (2001). A small part of this case study will be reproduced here to illustrate what is possible when the UML design process and Lean Cuisine+ models are combined.

The case study investigates the software application for a university timetabling system. Figure 4 shows the UML use case diagram for this system.

Textual descriptions of use cases are recommended by the Unified Process and used by most UML-based methods. The description for the *Create Student Timetable* use case is provided in Figure 5.

A UML class diagram for the Timetable Viewer project is provided in Figure 6. It should be noted that this study is concentrating on aspects of user interface design and consequently Figure 6 provides only an *outline* of the class diagram for the Timetable Viewer; many details have been omitted.

Figure 4. UML use case diagram for the Timetable Viewer

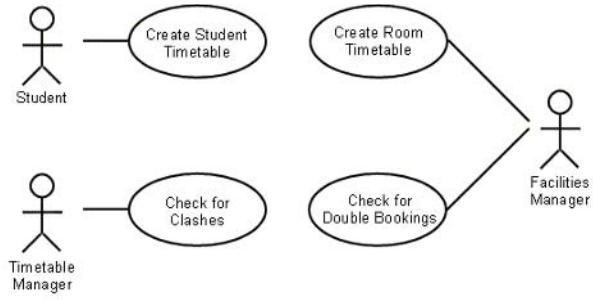


Figure 5. Textual description for the Create Student Timetable use case

Use Case:	*Create Student Timetable*
Actor:	Student
Goal:	The student wants to print out their own timetable.
Preconditions: None.	
Main flow:	The student selects a semester and then selects a number of courses. As each course is selected, the system includes the times for that course in the timetable display. The student can select to display only lectures, only tutorials, only laboratories or any combination of these. When the display is satisfactory, the student can print the timetable and the use case terminates.
Exceptions:	The student selects the wrong course. To correct the problem, the student may deselect the course. The system displays the timetable without the deselected course.
Postconditions: None.	

Figure 6. UML class diagram for the Timetable Viewer. This is not a complete class diagram.

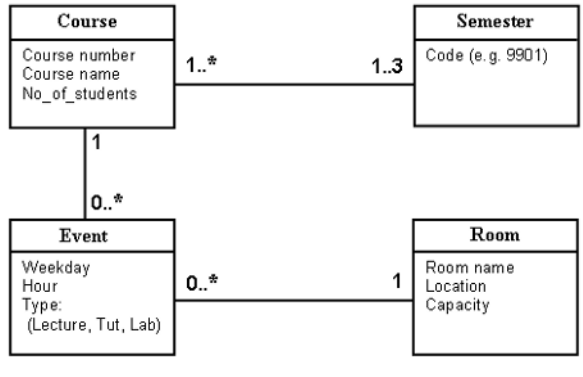
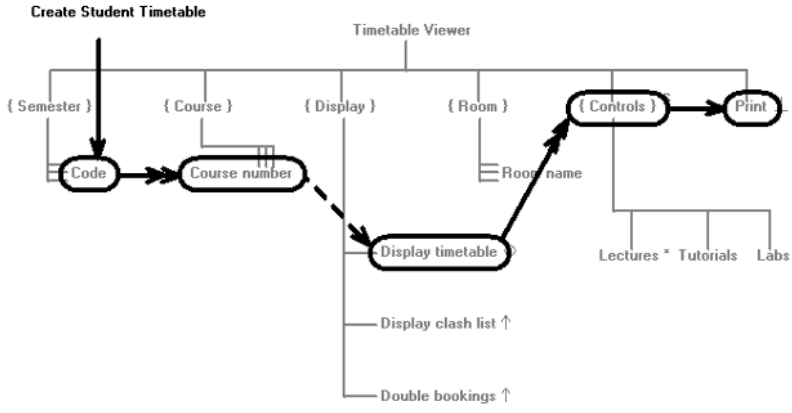


Figure 7. A Lean Cuisine+ task sequence for the Create Student Timetable use case



A Lean Cuisine+ model of the user interface is constructed, using the UML classes as a base for the tree structure. Each use case is then represented as a Lean Cuisine+ *task sequence* in the model. For example, see Figure 7.

It must be noted that this process is highly iterative and only the finished model is illustrated here.

Guidelines for Upgrading a User Interface

When upgrading an existing software application, there are two possible ways of handling the user interface — either (a) the essence (dialogue) of the interface can be retained and passed onto a new user interface, or (b) the old interface can be destroyed and a new one created.

In the case of (a), a Lean Cuisine+ model should be constructed that represents the dialogue and tasks of the existing interface. Ongoing research is investigating the possible automation of this process (Scogings & Phillips, 1998). In the case of (b), a Lean Cuisine+ model can be derived from the UML use cases and class diagrams for the application.

In both cases, new elements of the dialogue structure and new tasks can be added to the Lean Cuisine+ model, if required. Similarly, existing structure and/or tasks can be removed. The Lean Cuisine+ model can be analysed for completeness and efficiency and finally, a new user interface can be constructed from the Lean Cuisine+ model.

Summary

The user interface is a vital part of any software application, and may well determine its success or failure. Many software upgrades include a complete revision of the user interface. This process is greatly aided by the existence of an accurate model of the dialogue and tasks required of the new interface, which is independent of its surface "look and feel." Task and dialogue modeling are well established but have been largely carried out as separate activities. Dialogue notations (e.g., STDs, Petri nets) were not designed to model user interfaces and can be clumsy and difficult to construct and understand. The linkage between tasks and dialogue tends to be obscure. The user interface should be modeled as part of the overall software development process, but UML makes no specific provision for this.

The use of Lean Cuisine+ solves these problems by introducing a model of the user interface that combines both tasks and dialogue, and which combines naturally with UML. An existing interface can be modeled in Lean Cuisine+ and a new visible interface can then be constructed. The Lean Cuisine+ model can be constructed directly from UML use cases and class diagrams. During the process, the Lean Cuisine+ model can be checked for accuracy and efficiency.

References

Apperley, M.D., & Spence, R. (1989). Lean Cuisine: A low-fat notation for menus. *Interacting with Computers, 1*(1), 43-68.

Brooks, R. (1991). Comparative task analysis: an alternative direction for human-computer interaction science. In J. Carrol (Ed.), *Designing interaction: Psychology at the human-computer interface* (pp. 50-59). Cambridge: Cambridge University Press.

Church, K., & te Braake, G. (2002). The future of software development. In S. Valenti (Ed.), *Successful software reengineering* (pp. 99-110). Hershey, PA: IRM Press.

Collins, D. (1995). *Designing object-oriented user interfaces*. Redwood City, CA: Benjamin Cummings.

Conallen, J. (2002). *Building Web applications with UML*. Boston: Addison-Wesley Longman.

Constantine, L., & Lockwood, L. (1999). *Software for use*. Reading, MA: Addison-Wesley.

Eriksson, H-E., & Penker, M. (1998). *UML toolkit*. New York: John Wiley & Sons.

Jacquot, J-P., & Qesnot, D. (1997). Early specification of user interfaces: Towards a formal approach. *Proceedings of ICSE'97,* Boston, (pp. 150-160).

Janssen, C., Weisbecker, A., & Ziegler, J. (1993). Generating user interfaces from data models and dialogue net specifications. *Proceedings of INTERCHI'93* (pp. 418-423). ACM.

Kemp, E., & Phillips, C. (1998). Extending support for interface design in software engineering methods. *Proceedings of HCI'98*. Sheffield, UK, (pp. 96-97).

Kovacevic, S. (1998). UML and user interface modelling. In J. Bezivin & P-A. Muller (Eds.), *Lecture Notes in Computer Science 1618*. New York: Springer-Verlag.

MVC Design. (2003). *Re: MVC design or architecture pattern.* Available online at *http://www.mail-archive.com/gang-of-4-patterns@cs.uiuc.edu/msg00038.html*

Pawson, R., & Matthews, R. (2002). *Naked objects*. Hoboken, NJ: John Wiley & Sons.

Phillips, C. (1993). *The development of an executable graphical notation for describing direct manipulation interfaces*. PhD Thesis. Palmerston North, New Zealand: Massey University.

Phillips, C. (1995). Lean Cuisine+: An executable graphical notation for describing direct manipulation interfaces. *Interacting with Computers, 7*(1), 49-71.

Pinheiro da Silva, P., & Paton, N. (2003) User interface modelling in UMLi. *IEEE Software Magazine*, *20(*4), 62-69.

Quatrani, T. (2000). *Visual modelling with Rational Rose 2000 and UML*. Reading, MA: Addison-Wesley.

Roberts, D., Berry, D., & Isensee, S. (1997). OVID: object view and interaction design. *Proceedings of IFIP TC13 International Conference,* Sydney, Australia (pp. 663-664).

Rourke, C. (2002). Making UML the lingua franca of usable system design. *Interfaces* 50. British HCI Group. Swindon, UK.

Rumbaugh, J., Jacobson, I., & Booch, G. (1999). *The unified modelling language reference manual*. Reading, MA: Addison-Wesley.

Scogings, C., & Phillips, C. (1998). Beyond the interface: modelling the interaction in a visual development environment. *Proceedings of HCI'98,* Sheffield, UK (pp. 108-109).

Scogings, C., & Phillips, C. (2001). Linking tasks, dialogue and GUI design: A method involving UML and Lean Cuisine+. *Interacting with Computers, 14*(1), 69-86.

Chapter XIII

A Reuse Definition, Assessment, and Analysis Framework for UML

Donald Needham, United States Naval Academy, USA

Rodrigo Caballero, United Technologies Research Center, USA

Steven Demurjian, The University of Connecticut, USA

Felix Eickhoff, The University of Connecticut, USA

Yi Zhang, The University of Connecticut, USA

Abstract

This chapter examines a formal framework for reusability assessment of development-time components and classes via metrics, refactoring guidelines, and algorithms. It argues that software engineers seeking to improve design reusability stand to benefit from tools that precisely measure the potential and actual reuse of software artifacts to achieve domain-specific reuse for an organization's current and future products. The authors consider the reuse definition, assessment, and analysis of a UML design prior to the existence of source code, and include dependency tracking for use case and class diagrams in support of reusability analysis and refactoring for UML. The integration of these extensions into the UML tool Together Control Center to support reusability measurement from design to development is also considered.

Introduction

Software evolution addresses the changes a software system undergoes so as to remain, or become, a viable solution to a problem. Developing reusable software components is a critical aspect of software evolution since a reusable component offers the promise of being beneficial in situations unforeseen at the time of the component's development. Although reusable software components have been in use for over 30 years (McIlroy, 1968), their benefits, including reduced risk, limited development and maintenance costs, reduced time-to-market, increased quality and reliability, improved interoperability, and the support of rapid prototyping (*Software reuse executive premier*, 1996; Hall, 1999; Rine & Nada, 2000; Schmietendorf, 2000; Tsagias & Kitchenham, 2000), have proven difficult to demonstrate in practice. Today's component-based programming languages and their APIs support *actual reuse* of standard components (e.g., GUIs, communications, databases), but are less successful in attaining the *reuse potential* of components (Succi, 1995), a target of domain-and-organization-specific reuse (Meekel, 1997; Poulin, 1996) which represents a long-term investment in reuse (Sarshar, 1996). To support reuse potential, we provide an integrated reusability metric, framework, and tool (Price & Demurjian, 1997; Price, Demurjian, & Needham, 1997; Price, Needham, & Demurjian, 2001) with formal underpinnings (Caballero & Demurjian, 2002) for reusability definition, assessment, and analysis of software. We provide the ability to mark components/classes to indicate their reuse level, which can range from general (reuse in multiple contexts) to specific (single use). For example, in a retailing application, a supplier would have a general Item (reusable in many contexts), while an auto-parts supplier would have a less general AutoItem (reusable in that context), and individual retailers would have a specific WalmartItem (not reusable). Given a marking of an application's classes/components, our reuse metrics can objectively classify and measure dependencies (couplings) within and among classes/components, thereby identifying couplings that promote or hinder future reuse. This process can be automated with an algorithm for reusability assessment and refactoring (Caballero & Demurjian, 2002), and is supported by a Design Reusability Evaluation (DRE) tool (UConn, 2003).

To fully support reuse potential, a reusability definition, assessment, and analysis must be provided for the Unified Modeling Language (UML) *prior* to the development of software (code), to concentrate on reusing a design model and to monitor reuse as the model evolves when using UML via a tool (e.g., Rational Rose or Together Control Center). Specifically, this chapter details the integration of reuse and refactoring concepts for reusable UML components, allowing these components to be easily incorporated into a product family while tracking reuse assessments during design and following through to the development and maintenance stages. Our approach to reusability assessment and analysis in UML focuses on use cases and class diagrams. In practice, software engineers tend to start the process by defining use cases, followed by classes and the various activity views (sequence and collaboration diagrams), iterating across the various diagrams over time as the design evolves. Our approach supports the marking of use cases and classes with varying levels of generality, and then tracks and enforces this marking as a software engineer modifies the design. For example, as a software engineer starts with UML and defines use cases, the use cases are marked

with reuse levels that range from general to specific. As use cases interact with one another, and interact with new and existing classes as they are defined, our reuse model automatically notifies the software engineer when conflicts that hinder reuse are detected so that the UML components can be refactored. Further, as the design evolves, the reuse model tracks the couplings that are defined (via class, sequence, and collaboration diagrams) in order to assess the impact on reuse of existing components, which may result in the need to refactor. To provide a seamless environment for reuse definition, assessment, and analysis, our model has been integrated into the UML tool Together Control Center (TCC). The remainder of this chapter is organized as follows. First, we discuss foundational work and provide background information on our reuse and refactoring framework. Next, we present our reuse definition, assessment, and analysis framework for UML, providing reuse properties and associated refactoring guidelines. We then discuss the integration of the UML reuse framework into TCC. Finally, we present our conclusions and discuss our ongoing research.

Foundational and Related Work

Reuse metrics and models are used to estimate and reduce the software development effort. Frakes (1996) reviews six different types of reuse metrics and models. Our approach resembles what Frakes refers to as the "reusability" and "amount of reuse" models, as we focus on identifying the reuse potential of software as well as suggest ways to improve that potential. An initial attempt to develop formal metrics for object-oriented (OO) design (Chidamber, 1994) characterizes important properties for OO designs and mentions several candidate metrics to facilitate OO development while providing an evaluation of each candidate's metrics. Virtanen's (2000) component reuse metrics focus on estimating the component-based development, and calculates the time required based on excluding human effects, deriving from historical data of previous developments. Our approach is similar in that we evaluate the current product and recommend changes based on our metrics with the objective of improving software reusability. While Virtanen estimates the development time of a component based on historical data, we focus on reducing future development time by developing components with a high reuse potential. In Basili's (1990) studies on the Ada programming language's reusability, cluster analysis is carried out to identify couplings between data bindings. Based on this information, components are classified as either reusable or not. Basili's view is similar to our approach, except that we have the notion of classifying components as general or specific by the software engineer, before the coupling information is calculated. Moreover, couplings between independent components do not necessarily result in low reuse potential as long as the software engineer identifies these components as related, that is, as components intended to be reused together.

Component-based software engineering (CBSE) focuses on constructing large-scale software applications from previously existing components. In support of CBSE, toolkits such as Carnegie Mellon University's Aesop System (Garlan, 1995) provide environments tailored to a given set of designer-specific styles that guide developers in creating

architectural designs for their systems. Such toolkits provide mechanisms through which developers can mitigate disparities between assumptions made about a reusable component and the system in which the component is to be reused. Our work is similar in that we provide a formal framework for the reusability assessment of development-time components and classes via metrics, refactoring guidelines, and algorithms. However, we examine reusability from a domain-and-organization-specific perspective rather than focusing on collections of architectural styles generic enough to apply across domains. An international workshop on component-based software engineering held in Kyoto, Japan (Brown, 1998) identified two important component relationships aspects, those between components and object technology and those between components and software architecture, and stressed the importance of a clear definition of reuse from an organizational perspective. Our work addresses the latter relationship, between components and software architecture, and supports the tailoring of reuse at the organizational level.

Another aspect of component-based models for reuse involves reuse contracts (Steyaert, 1996; Cernuda, 2001; Lucas, 1997), which seek certification that the components in question are in fact reusable. Reuse contacts focus on the interfaces between components, offer guidelines for reuse within a specific problem domain, and can include an acquaintance clause which indicates that one component has knowledge about another component. Like the acquaintance clause of a reuse contract, our approach allows for an indication of the anticipated relationships between components. Our work builds upon this area to include a set of metrics through which acquaintance-like relationships may be evaluated, and offers refactoring guidelines that assist in determining how such relationships can be restructured to enhance the reuse potential of the components. Our work is similar to Cernuda's (2001) notion of a bypass, in which relationships between classes in an inheritance hierarchy are replaced with relationships closer to or further from the hierarchy's leaf nodes. We further this approach by providing refactoring rules that guide a designer in moving relationships both up and down an inheritance hierarchy, and provide an analysis of the impact on reuse that such design decisions encompass.

Background: A Reuse Model

In this section, we review our component reuse framework proposed in Price and Demurjian (1997), extended in Price et al. (2001), and formalized in Caballero and Demurjian (2002), based on two characteristics: class generality assessment and relations among classes. To summarize our previous work, we developed a methodology for assessing the reuse potential of components through which classes may be marked as either general or specific. A *general* class (G) is one that is expected to be reused in future applications, while a *specific* class (S) is only intended for one application. For example, an Item class for products is general to all retail applications, while a WalmartItem class descendant has store specific characteristics. In a later effort by Price et al. (2001), the model was extended to accommodate levels of generality to better exploit the reuse potential of classes, and led us to several definitions through which to base our dependency analyses of software designs. These definitions are as follows:

Definition 1: The generality level of a class C_i is denoted by G_i, where $G_i = 0$ is the most general class and $G_i = N$ ($N > 0$), when C_i is the most specific class.

For example, generalities for classes range from Item (G_{item}=0) to DeptStoreItem ($G_{deptStoreItem}$=1) to DiscountStoreItem ($G_{discountStoreItem}$=2) to WalmartItem ($G_{walmartItem}$=3), demonstrating four different levels. *Relations among classes* characterizes classes that are expected to be reused together in a future application, for example, an ItemCollection class (set of all Items) is related to the Item class. More often than not, related classes are coupled, leading to:

Definition 2: Class C_p is related to class C_q if C_p must use C_q in all future reuse. Related classes have dependencies in the direction of the relation in the form of attribute inclusion, method invocation, and object inclusion.

Definition 3: Each time any member of class C_p references any member of class C_q, a Coupling is said to exist from C_p to C_q. Couplings among unrelated classes prohibit reuse, while couplings among related classes may improve, hinder, or have no influence on the reuse of the classes.

It is important to note that an inheritance relation between two classes is an instance of a coupling. Specifically, if class C_p extends class $C_{p'}$, then class C_p is coupled to class $C_{p'}$. If class C_p is reused in an application, then class $C_{p'}$ must also be brought along for the new application to function properly. Class generality and relations among classes are an important part of our framework for reusability analysis, clarified by the following properties:

Property 1: In an inheritance hierarchy, the parent class is equally general or more general than its children, meaning that children have at least the reusability level of the parent, plus optional specific (less general) attribute(s) and/or method(s).

For example, WalmartItem is less general than its ancestor Item due to WalmartItem's additional, specialized class members. An interesting case arises when subclasses override methods defined in the superclass. Consider the Java classes ClassX and ClassY defined as follows:

```
public class ClassX {
  public int f(int i) {
     //Very specific code here
  }
}
public class ClassY extends ClassX {
  public int f(int i) {
     return i;
  }
}
```

Observe that although class ClassX is a super class of ClassY, the implementation of class ClassX is more specific than the implementation of class ClassY. In particular, the implementation of method ClassX.f(int i) might contain couplings to classes marked Specific. However due to Property 2, the generality of ClassY is at least the generality of class ClassX due to the coupling induced by the inheritance relationship.

Property 2: The generality level of a class is equal to the generality level of the least reusable related class, as considered over all the classes that a class is related to. Given two classes, C_p and C_q with generality levels G_p and G_q there are two cases: C_p is more specific than C_q ($G_p > G_q$), in which case C_p is the least reusable class; and C_p is more general than C_q ($G_p < G_q$). If there is a relation from C_p to C_q, C_p may be dependent on specific features of C_q that will condition the reusability of C_p to situations when C_q is reusable, that is, the generality level of C_p must be equal to or greater than the reusability level of C_q.

Property 3: Couplings between unrelated classes are undesirable and hinder reuse since unrelated classes are not intended to be reused together in future applications. If there are dependencies among unrelated classes, the coupled classes must be brought along to the reusing application in order for the system to function properly, which contradicts the intent of not reusing the components together.

Software engineers label or mark their classes with a generality level, which denotes the developer's reuse expectancy for the class. To simplify, assume that there are only two levels of generality, G and S; if there are multiple levels, as is the case with our model, one proceeds in a similar manner, focusing on only two levels at a time. Between any two generality levels there are four types of couplings: a G class can depend on another G class, a G class can depend on a S class, a S class can depend on a G class, and a S class can depend on another S class. These dependencies can take the form of a method call, inheritance, or an instance variable reference. As classes also may be related or unrelated, we have identified a total of eight separate types of couplings (Price and Demurjian, 1997; Price et al., 1997; Price et al., 2001):

Type 1. G → G among related classes is an asset to reuse, the two classes are expected to be reused together, and the objective is to increase these couplings.

Type 2. G → G among unrelated classes is undesirable since the source and destination are not expected to be reused together. Refactor by moving both the source and destination to Specific descendant classes or making the classes related.

Type 3. G → S among related classes is undesirable, since the General class (to be reused) depends on a class which is not expected to be reused. Refactor by moving the source to a Specific descendant or the destination to a General ancestor.

Type 4. G → S among unrelated classes is undesirable (source expected to be reused, destination is not). Refactor by moving the source to a Specific descendant class.

Type 5. S → G among related classes does not hinder reuse since the source of the coupling is not expected to be reused at all. Refactor to improve reuse by moving the source to a General ancestor.

Type 6. S → G among unrelated classes does not hinder reuse since the source of the coupling is not expected to be reused. There is no need to refactor in this case.

Type 7. S → S among related classes does not hinder reuse since the source of coupling is not expected to be reused. Refactor to improve reuse if both source and destination are moved to their General ancestors.

Type 8. S → S among unrelated classes represents the desired situation for couplings between unrelated classes; they need to be among the Specific classes.

Given the eight different coupling types, we refactor to improve reuse. Our method is based on the reduction of couplings that hinder future reuse, by transitioning couplings to types that promote or are neutral to future reuse. For example, by identifying and transitioning a coupling from Type 3 to Type 1, we are able to move a coupling that hinders reuse (Type 3) to one that promotes reuse (Type 1).

Reuse Definition, Assessment, and Analysis in UML

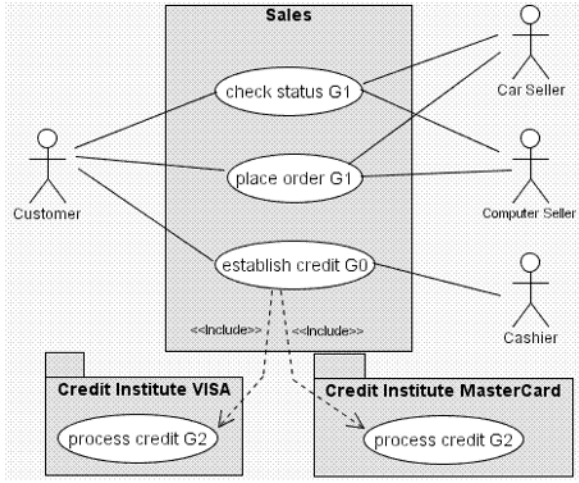
In this section, we examine reuse definition, assessment, and analysis in UML (Booch, Jacobson, & Rumbaugh, 1999) from the perspective of use cases, classes, behavior modeling, and component diagrams. We concentrate on the steps that software engineers take as they create and evolve their design, and we provide properties and associated refactoring guidelines to facilitate the improvement of reuse potential. While the following discussion is sequential, a software engineer would typically iterate among the various UML diagrams during design.

Use Case Diagrams

Use case diagrams are made up of three different elements: actors, systems, and use cases (see Figure 1). A UML system collects connected units that are organized to accomplish some purpose, for example, the Sales system in Figure 1. From a reuse perspective, we are interested in the couplings between systems. In Figure 1, to reuse Sales, one must also use the Credit Institute Visa and the Credit Institute MC systems. Conversely, Credit Institute Visa or Credit Institute MC can be reused without reusing Sales. Currently, reuse at the system level is not supported; our focus is on understanding the reusability of the use cases so that when a use case is reused, we only reuse what is necessary for supporting that use case. Actors, as shown in Figure 1, represent a set of roles for interacting with use cases. Assigning generalities to actors does not make sense, since actors are external entities and are not part of the system under consideration.

Use cases provide us with the first UML construct to begin to define a reuse definition, assessment, and measurement framework for UML. In the initial stages, as a software

Figure 1. An example use case diagram



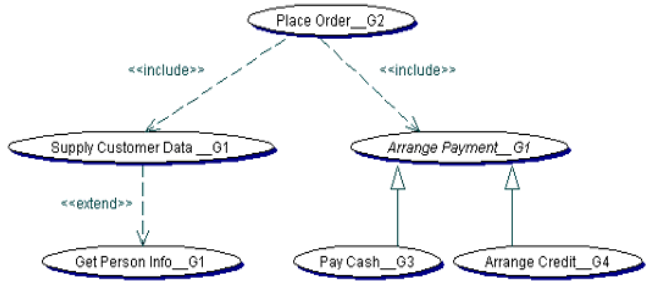
engineer defines use cases, the generality level can be assigned in a similar fashion to classes (Price and Demurjian, 1997; Price et al., 1997; Price et al.,2001). The assigned generality level of a use case measures its reuse potential, and allows comparison against other use cases, in an effort to track the couplings among use cases.

Recall the properties presented in the Background Section that delineate reuse definition requirements: Property 1: generality of a parent class versus its child classes; Property 2: generality of a class is equal to generality of the least reusable coupled class; and Property 3: unrelated classes with dependencies between them that hinder reuse. To augment these properties, we define additional properties that focus on the reusability requirements for UML that must be enforced by the reuse framework in order to "design" reusable UML artifacts. At initial design stages, with only use cases defined, the emphasis on reuse definition and analysis is primarily on the assignment of generality level, and the impact of these assignments by the different types of relations between use cases in UML, namely <extend>, <include>, and <generalization>.

Relations between use cases are transitive in the direction of the relation arrows. An exception is when a use case is marked as abstract, and the generalization links are followed against the arrow direction. The first link in a relation chain sets the type of the relation to all transitive related use cases (if the first link is an <include>, all transitive related use cases are included). Transitivity of relations, as shown in Figure 2, is important since it allows the tracking of indirect dependencies that can hinder reuse.

Figure 2 depicts the following relations: "Place Order" includes: "Supply Customer Data," "Get Person Info," and "Arrange Payment." As "Arrange Payment" is abstract, it also includes "Pay Cash" and "Arrange Credit"; "Supply Customer Data" extends "Get Person Info"; "Pay Cash" inherits from "Arrange Payment"; and "Arrange Credit" inherits from "Arrange Payment." Considering use cases UC_A and UC_B with respective generalities $G_{UC\text{-}A}$ and $G_{UC\text{-}B}$ leads us to the following additional properties:

Figure 2. Transitivity in use case relations



Property 4. UC_A extends UC_B is a relation in which UC_A adds behavior to UC_B, meaning that UC_A is at most as general as UC_B.

Property 5. UC_A includes UC_B is a relation of the behavior sequence of the supplier UC_B into the interaction sequence UC_A. From a reuse perspective, it means that UC_A is at most as general as UC_B, for example, in Figure 2 use case "Supply Customer Data" is more general or equally general than use case "Place Order."

Property 6. UC_A generalizes UC_B is a relation of a specialized UC_B to a more general UC_A, meaning that UC_B is at most as general UC_A.

Violating these properties yield bad couplings (i.e., poor reusability) between the use cases which must be corrected. This is facilitated by our development of refactoring guidelines (RGs) for software engineers, which enumerate the possibilities available for correcting the violations. For Properties 4, 5, and 6, RG1, RG2, and RG3, respectively, are:

RG1, RG2: To enforce Property 4 or Property 5, the refactoring rule is: If $G_{UC\text{-}B} > G_{UC\text{-}A}$, then refactor by making UC_B more general (or UC_A more specific) so $G_{UC\text{-}B} \leq G_{UC\text{-}A}$ or by removing the extend/include.

RG3: To enforce Property 6, the refactoring rule is: If $G_{UC\text{-}A} > G_{UC\text{-}B}$, then refactor by making UC_A more general (or UC_B more specific) so that $G_{UC\text{-}A} \leq G_{UC\text{-}B}$ or by removing the generalization.

The use-case-to-use-case reuse assessment applies our refactoring guidelines from the Background Section. When the generalities differ by one level, the refactoring guidelines for Types 1, 3, 5, and 7 apply. When the generalities differ by two or more levels, additional refactoring guidelines (Price et al., 2001) apply. The change of generality of either, or both, of the two use cases may result in conflicts with other use cases that are connected to UC_A and UC_B via extend, include, or generalize relationships. Use cases that are not related at initial design stages do not have dependencies, since they are not linked by extend, include, or generalize. Unrelated use cases with dependencies to classes that are not related to the use case must be checked, tracked, and corrected.

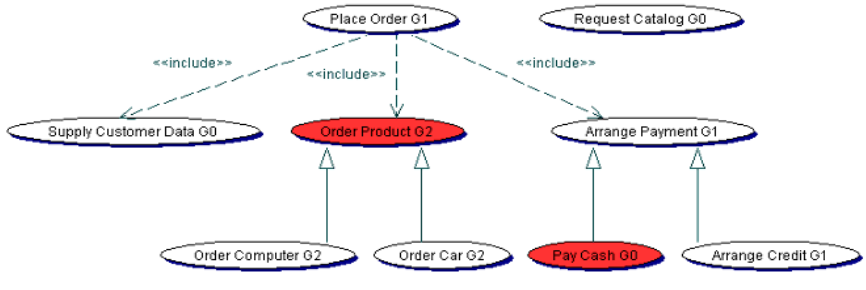
Figure 3 gives a use case diagram with the generality level of each use case assigned. In Figure 3, the level of generality of each use case follows the name of the use case. As currently marked, use case "Order Product" violates Property 5 for <include>, by being more specific than use case "Place Order," and use case "Pay Cash" violates Property 6 for <generalize>, by being more general than "Arrange Payment." Refactoring the design in Figure 3 is essential for maintaining the reusability dependencies that adhere to Properties 1 to 6, in order to reuse use cases in future domain-and-organization-specific applications. By using RG2, "Place Order" <includes> "Order Product" can be refactored by: making the generality of "Order Product" G_1 (more general), by making the generality of "Place Order" G_2 (more specific), or by removing the <include>.

The selection amongst these three choices is at the discretion of the software engineer. Similarly, by using RG3, the generality of "Pay Cash" can be changed to G_1, the generality of "Arrange Payment" can be changed to G_0, or the generalization between the use cases can be deleted. For the purposes of the discussion, assume that both "Order Product" and "Pay Cash" are changed to G_1.

Class Diagrams

Once the use cases have been initially defined, the software engineer can begin to model classes. The reuse potential of class diagrams is evaluated according to our earlier metrics (Price & Demurjian, 1997; Price et al., 1997; Price et al., 2001) that measure class-to-class couplings. This effort assumes the existence of detailed couplings, such as method calls and attribute inclusions, which will not be the case at initial stages of the design process. Rather, we assume that a software engineer is defining use cases and classes (initially by name only), assigning generality levels to use cases and classes, and establishing relations between use cases and classes, and among classes, all of which have the potential to impact reuse. In a UML design, there can be relations between use cases and classes as follows:

Figure 3. "Order Product" and "Pay Cash" violate Properties 5 and 6



Definition 4: A use case UC_A is *related to* a set of classes $\mathfrak{C}_A = \{C_1, C_2, \ldots, C_n\}$ for some n, if UC_A relies on $\mathfrak{C}_A$ for its functionality. It is left to the developer of the application to determine which classes implement the use case and belong to $\mathfrak{C}_A$.

Reusing UC_A means reusing all classes in the set $\mathfrak{C}_A$. The □ sets for the example are:

$$\mathfrak{C}_{Place\ Order} = \{Order\ Container,\ Order,\ ItemDB\ \},\ \mathfrak{C}_{Order\ Product} = \{Item\ \},$$
$$\mathfrak{C}_{Order\ Computer} = \{Computers\ \},\ \mathfrak{C}_{-\ Order\ Car} = \{Cars\ \},\ \mathfrak{C}_{Request\ Catalog} = \{Catalog\ \},$$
$$\mathfrak{C}_{Supply\ Customer\ Data} = \{Customer,\ DB\ Access\ \},\ \mathfrak{C}_{Arrange\ Payment} = \{Payment\}$$

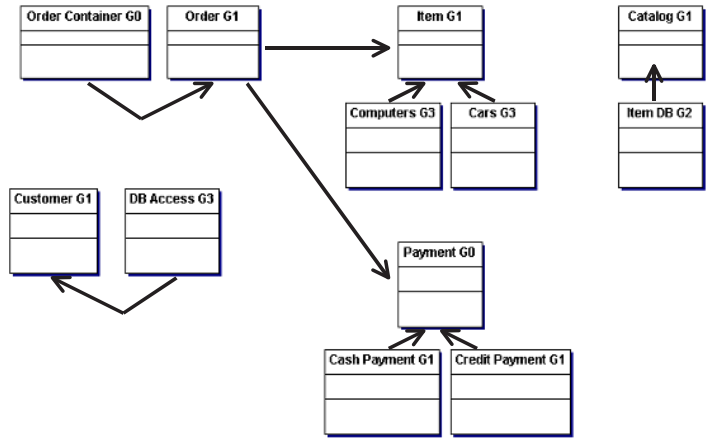
Given this definition, we can now examine use case generality when a use case is related to a set of classes, assuming that both the use cases and the classes related to them have been assigned generality levels. The software engineer can then redefine the generality level of any class as part of an iterative design process. Intuitively, a class *C* is at least as general as any use case *UC* related to it, or stated another way, a use case *UC* is no more general than the most specific (least general) class *C* to which it is related, which leads to the Property 7 and guideline RG4.

Property 7. Suppose that UC_A is related to a set of classes $\mathfrak{C}_A = \{C_1, C_2, \ldots, C_n\}$ for some n. Then, the generality of UC_A must be as specific as the most specific class in $\mathfrak{C}_A$, and may be more specific, that is, $G_{UC\text{-}A} = \max\{\text{generality } \forall C_i \in \mathfrak{C}_A\}$.

RG4: To enforce Property 7, the refactoring rule is: generality change of UC_A or one or more $C_i \in \mathfrak{C}_A$ until $G_{UC\text{-}A} = \max\{\text{generality } \forall C_i \in \mathfrak{C}_A\}$, or the removal of all classes in $\mathfrak{C}_A$ that cause the $G_{UC\text{-}A} = \max\{\text{generality } \forall C_i \in \mathfrak{C}_A\}$ to be violated. The changes made in this situation may impact elsewhere, and our tool (see Prototyping in Together Section) automatically detects and suggests alternatives for correction of the software design.

Figure 4 contains a class diagram with generality levels annotated next to each class name. Note that while we have shown the various relations among the classes, we have not identified the type of each relation (e.g., dependencies, associations, and generalizations) since it is not needed for our purposes. Figure 4 contains two violations of Property 7 as a result of dependencies among classes: Use case "Place Order" is related to a more specific class, "Item DB," than itself; and Use case "Order Car" is related to a more specific class, "Cars," than itself. These can be refactored according to RG4, respectively, as: Change "Item DB" to be more general with a level G_1, "Place Order" to be more specific with a level of G_2, or remove "Item DB" from "Place Order"; and Change "Cars" to be more general with a level G_1, change "Order Car" to be more specific with a level G_3, or remove "Cars" from "Order Car." Since changing relations between use cases and classes also may affect the generality levels elsewhere in the design, the reuse must

Figure 4. Sample class diagram with generalities



be constantly reanalyzed. The prototype in the Prototyping in Together Section of this chapter supports this through warnings and alerts.

In addition to Property 7, we also must track dependencies that can impact reuse among use cases that are related and unrelated, and between use cases and classes. *Use-case-to-use case dependencies* involve both related (expected to be reused together) and unrelated (not expected to be reused together) situations. The *use-case-to-use-case dependencies related* situation, as given in Property 8 and RG5, assumes that there are two use cases that are related to one another, which must be mirrored by a relation at the class level.

Property 8. Suppose that UC_A is related to $\mathfrak{C}_A$, and UC_B is related to $\mathfrak{C}_B$. If UC_A is related to UC_B (extend, include, or generalize — see Reuse Section), then there has to be at least one transitive relation chain from one $C_i \in \mathfrak{C}_A$ to one $C_j \in \mathfrak{C}_B$.

RG5: To enforce Property 8, the refactoring rule is: add one or more dependencies between class(es) $C_i \in \mathfrak{C}_A$ and class(es) $C_j \in \mathfrak{C}_B$, or remove the relation between UC_A and UC_B.

RG5 maintains dependencies among use cases to allow their reuse in future settings. In the *use-case-to-use-case dependencies unrelated* situation, given in Property 9, since the two use cases are not related, any class dependency between them requires an alert to the software engineer to correct the situation following RG6.

Property 9. Suppose that UC_A is related to $\mathfrak{C}_A$ and UC_B is related to $\mathfrak{C}_B$, and that UC_A is not related to UC_B (i.e., there is *not an* extend, include, or generalize relation). Then, if at least one $C_i \in \mathfrak{C}_A$ is directly related (i.e., not by a transitive relation chain) to at least one $C_j \in \mathfrak{C}_B$, there must be a refactoring.

RG6: To enforce Property 9, the refactoring rule is: make UC_A related to UC_B or remove all dependencies between all $C_i \in \mathfrak{C}_A$ and all $C_j \in \mathfrak{C}_B$.

The *use-case-to-class dependencies* situation involves a use case related to a class, which in turn is related to another class which is not in the $\mathfrak{C}$ set of the use case. To satisfy reuse requirements, a use case must be related to all other classes, both direct (in $\mathfrak{C}$) and inferred (via a relation). This situation is handled by Property 10/RG7 as described.

Property 10. Suppose that UC_A is related to $\square_A$. Then for all $C_i \in \mathfrak{C}_A$, if there exists a transitive relation chain from C_i to some $C_j \notin \mathfrak{C}_A$, then UC_A must also be related to C_j in some manner. UC_A can be either related directly to C_j, or it can be related to some UC_B, to which C_j is related.

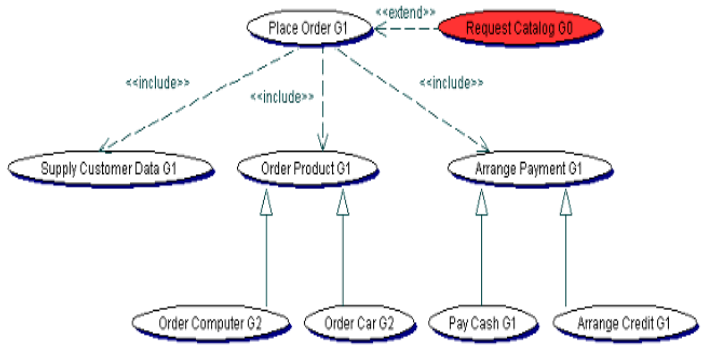
RG7: To enforce Property 10, the refactoring rule is: $\forall C_i \in \mathfrak{C}_A$ related to some $C_j \notin \mathfrak{C}_A$ include C_j in $\mathfrak{C}_A$ ($\mathfrak{C}_A = \mathfrak{C}_A \cup C_j$), or relate UC_A to some UC_B where $C_j \in \mathfrak{C}_B$, or unrelate UC_A to C_i ($\mathfrak{C}_A = \mathfrak{C}_A - C_j$). Note that this refactoring guideline maintains dependencies among use cases to allow their reuse in future settings.

To explore Properties 8 to 10, and RG5 to RG7, we continue our example as established in Figures 2 and 3, with the $\mathfrak{C}$ sets as given after Definition 3. There are two violations of Property 8 and 10, which can be refactored according to RG5 and RG7, and one problem with the design corrected with additional relations:

- Use case "Place Order" in Figure 3 includes use case "Supply Customer Data," but there are no relations between $\mathfrak{C}_{Place\ Order}$ and $\mathfrak{C}_{Supply\ Customer\ Data}$ in Figure 3. Use RG5 to add a relation from class "Order" to class "Customer" in Figure 3.
- Use case "Place Order" is related to class "Item DB," and "Item DB" is related to class "Catalog." "Place Order" should therefore be related to "Catalog" or some use case that is related to it (like "Request Catalog"). Use RG7 to add an <extend> relation from "Place Order" and "Request Catalog" in Figure 5. Note that with this action, Property 5 is now violated; since use case "Request Catalog" extends use case "Place Order" it should be at least as specific as "Place Order." Our prototype (See Prototyping in Together Section) automatically tracks these new violations.
- The software engineer should consider relating classes from Figure 4 to use case "Pay Cash" and use case "Pay Credit" in Figure 2, as these use cases are not yet related to any classes. Add relations from use case "Pay Cash" to class "Cash Payment" and from use case "Arrange Credit" to class "Credit Payment."

Refactoring using RG5 to RG7 to maintain Properties 8 to 10 is not a trivial task, and requires a deep understanding of application content and semantics in order to be successful. Relations can be added or removed at will; but relations should only be

Figure 5. Refactored use case diagram of Figure 3



changed if they make sense for the system. A reuse-focused UML tool can identify situations that require change, but the software engineer must use their own knowledge and experience and understanding of the domain to make appropriate changes.

As software engineers start to add details to their designs, including attributes, member functions, associations, and inheritance, there is an impact on reuse. New attributes and member functions all raise the possibility of new dependencies between classes that impact reuse. An attribute or member function argument whose type is that of another class may reveal a dependency that impacts Properties 8, 9, or 10, which would need resolution via RG5, RG6, or RG7. For example, in Figure 3, suppose an attribute "Last Payment Made" of type "Payment" is added to class "Customer," which is a new dependency.This new dependency has a direct impact on use case "Supply Customer Data," since $\mathfrak{C}_{Supply\ Customer\ Data} = \{Customer, DB\ Access\}$ does not contain "Payment," which violates Property 10, and must be corrected by RG7. Associations and inheritance also may impact properties. For example, defining a new association or inheritance between classes can impact Properties 8, 9, or 10, in a similar fashion to the example given for attributes. For new inheritance relations, a derived class cannot be more general than its parent (Property 1).

Finally, as the design continues to evolve with the development code, the actual count of couplings between classes can be more carefully tracked, which is supported in our current reuse framework (Caballero & Demurjian, 2002; Price & Demurjian, 1997; Price et al., 1997; Price et al., 2001) and the design reusability evaluation (DRE) tool. Coupling counts help the software engineer make reusability-related decisions about the overall design. For example, if the coupling count for one class to another is high and there is no relation defined between them, the engineer might want to add a relation showing that these components are intended to be used together. On the other hand, if the coupling count is low, the engineer might consider changing his design such that the classes are not dependent. As a UML design evolves toward code, the transition to the DRE tool is seamlessly handled (see Prototyping in Together Section).

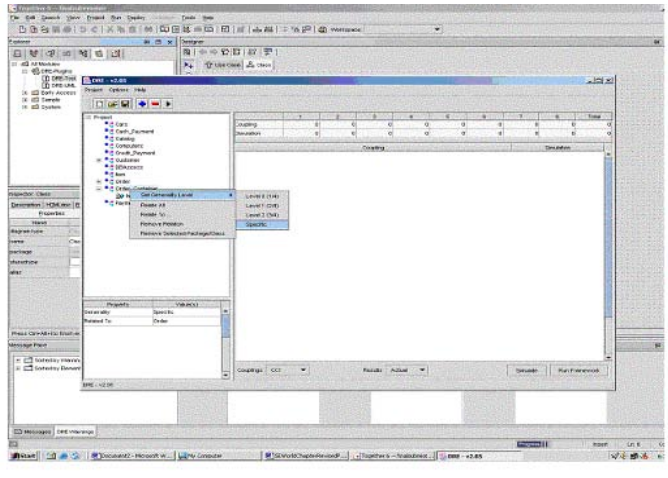
Prototyping in Together Control Center

In this section, we describe the ongoing prototyping of our original reuse framework (Price & Demurjian, 1997; Price et al., 1997; Price et al., 2001) and the extensions to support reuse of UML into Together Control Center, TCC (TogetherSoft, 2003). The DRE tool measures and reports the reuse potential of Java software according to our original reuse framework (Price & Demurjian, 1997; Price et al., 1997; Price et al., 2001) and extensions for automated refactoring (Caballero & Demurjian, 2002). The software engineer analyzes code by specifying the two reuse characteristics of the classes that comprise the application: the generality and the relations between them. DRE analyzes the source code of an application and tabulates the coupling counts for the eight coupling types we previously discussed in the Background Section of this chapter. A report of these coupling counts is presented to the user, along with refactoring guidelines for improving couplings that hinder future reuse. The DRE prototype can be downloaded at UConn (UConn's Reuse Web site).

As shown in Figure 6, we have integrated DRE into Together Control Center (TCC) with a focus on reuse assessment of the software produced using TCC, rather than based on UML designs. Our use of TCC offers us a selection of software audits and metrics that a software engineer can employ to measure different aspects of code. We integrated our reuse framework and metrics as a user-defined metric to be consistent with TCC.

TCC provides a rich set of Open APIs and a plug-in structure that has facilitated the integration of DRE and our reuse framework into TCC. Using TCC's OpenAPIs, functions, and objects, we developed a bridge between our standalone DRE application and the TCC editor, providing the ability to apply our reuse evaluations to any TCC Java project. A TCC property window pane for classes was extended, as shown in Figure 6, for the generality and relation information of each of the classes. This information can

Figure 6. Running the design reusability evaluation (DRE) tool in TCC



be accessed and updated through the standard Together-defined window to view property information.

Once the generality levels of classes and relations among classes have been established by the software engineer, our reuse metric can be run via TCC by activating the standard DRE tool to parse through the source files (stored under the control of TCC) and determine and classify all couplings that exist. While the standard metrics and audits of TCC display themselves in a TCC results pane, the complete DRE tool opens as a new application for use by the software engineer. Changes made to source code within the DRE environment propagate back to TCC, and vice versa, providing round trip engineering with respect to reuse (Figures 6, 7, and 8). The generalities of classes and relations among classes are persistently stored into Javadoc-style comments for the methods/data members, which can be parsed by DRE, so that future reuses of classes can automatically initialize the generalities/relations.

Figure 7. Round trip engineering

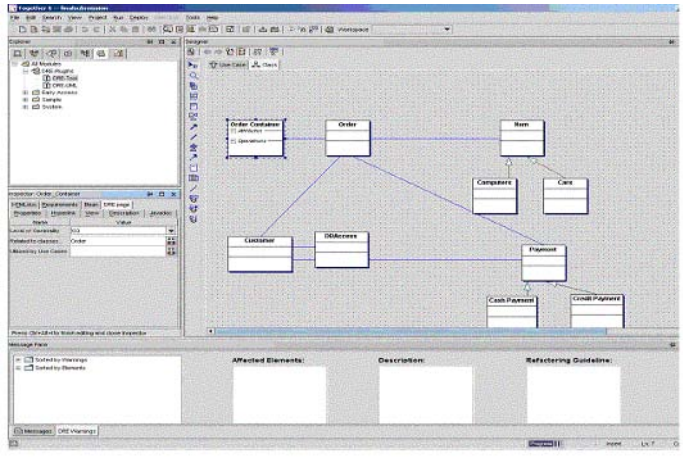
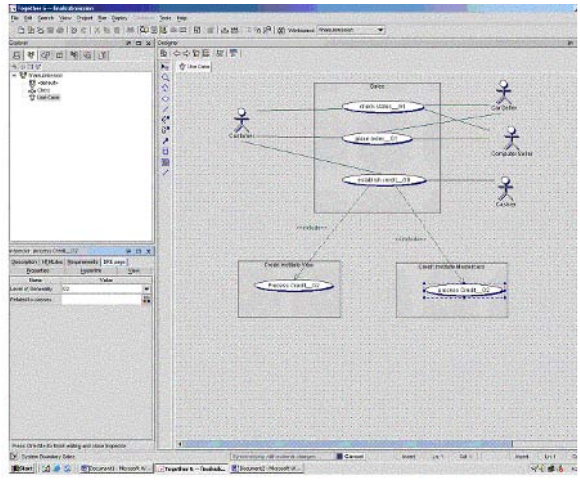


Figure 8. Creating use cases and assigning generalities



In addition, our reuse definition, assessment, and analysis framework for UML has been integrated into Together Control Center (TCC), providing direct support to the properties and refactoring guidelines for UML designs in order to allow software engineers using TCC to define, track, and manage reuse at all stages of design and development. This is inclusive from initial design through detailed design and implementation, and is supported by the integration of DRE and TCC. The TCC is a full concept-to-code CASE tool, allowing the incorporation of our reuse framework methodology to span the entire software life cycle.

The early support in the design process of a UML reuse framework as presented in the Reuse Section allows: setting generalities for the assessment and analysis resulting in the enforcement of Properties 4 to 13; and presenting to the software engineer the refactoring guidelines RG1 through RG10 that improve future reuse. As use cases are

Figure 9. Violations and refactoring guidelines

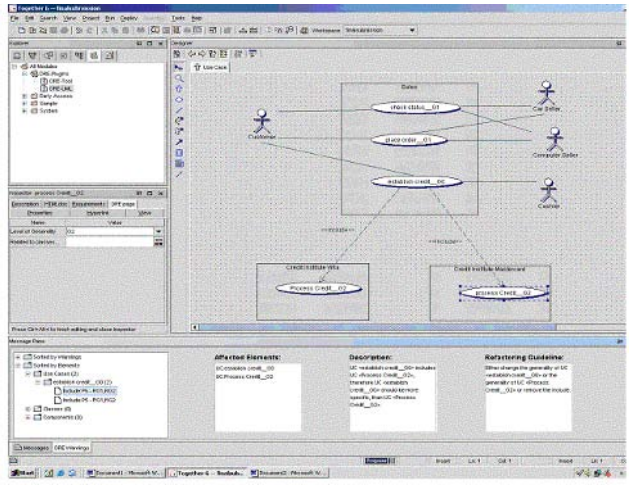
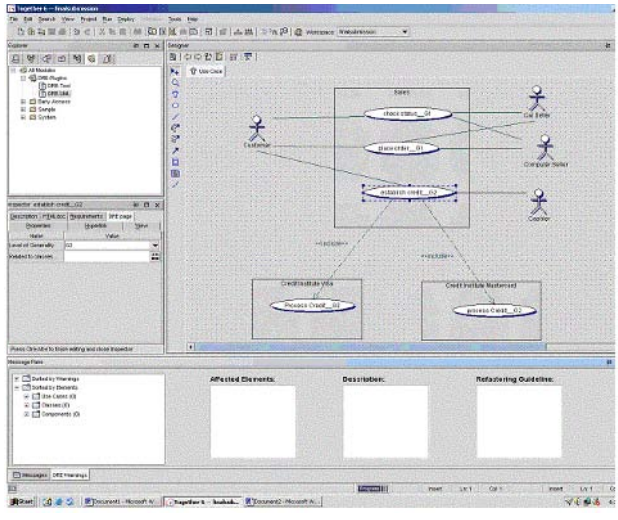


Figure 10. Corrective measures and results



created and marked with a generality level by the software engineer (see Figure 8) our extended TCC enforces Properties 4, 5, and 6. TCC alerts the software engineer that properties have been violated, and provides refactoring guidelines that suggest changes to improve the reusability of the design (RG1, RG2, and RG3) (see Figure 9).

Following these guidelines, the software engineer can take corrective measures and improve the design from a reuse perspective (see Figure 10). The generality markings are stored in TCC, allowing the user to retrieve and/or change the markings in the future. The TCC use case properties were extended to support use case generality level marking and relations among use cases. The properties windows where use case generality level and relations among use cases are analogous to the class properties windows illustrated in Figure 11. As use cases are defined and related to one another (see Figure 4), our extension to TCC enforces Properties 4, 5, and 6. If these properties are violated, TCC alerts the software engineer and provides reuse guidelines RG1, RG2, and RG3 as suggestions to improve the reusability of the design.

As an application's design continues to evolve, the software engineer will follow the use cases with the creation of class diagrams to give application functions more detail. The software engineer employs the class diagram property windows as given in Figure 11 to define both the generality and relations among classes. After these class diagrams are created and marked, the use cases that comprise an application can be related to the classes (via Definition 3 in the reuse section) as illustrated in Figure 12. At this stage of design, the generality of the use case versus the generality of the classes related to the use case is enforced with refactoring via RG4 as needed. Also, Properties 8, 9, and 10, for use-case-to-use-case and use-case-to-class dependencies, and Properties 11, 12, and 13, for components, are tracked, enforced, and refactored (via RG5 to RG10) (Figure 13).

To track and analyze considerations for all properties, a reuse assessment pane for TCC has been prototyped, as shown in Figure 14. There are four windows in Figure 14 that track the current warnings and violations (first window), the affected elements for a selected warning (second window), and for a particular selectable element, a description of the problem (third window), and the refactoring guidelines (fourth window).

This organization is consistent with the properties and refactoring guidelines as presented in Reuse Section. Using these various windows, the software engineer can explore the reuse problems with a UML design, make changes, and analyze their impact.

Conclusion and Ongoing Work

Developing reusable software components is a key aspect of software evolution. Evaluating the *reuse potential* of components at design time can benefit from an integrated reusability tool that provides an assessment of software design decisions. We have presented such a reuse definition, assessment, and analysis framework for UML that helps the software engineer to increase reusability since early stages in the development process. We focused on future reuse, extending our previous work focused at reusability at the class level, to consider reusability of UML use case diagrams.

Figure 11. Properties windows

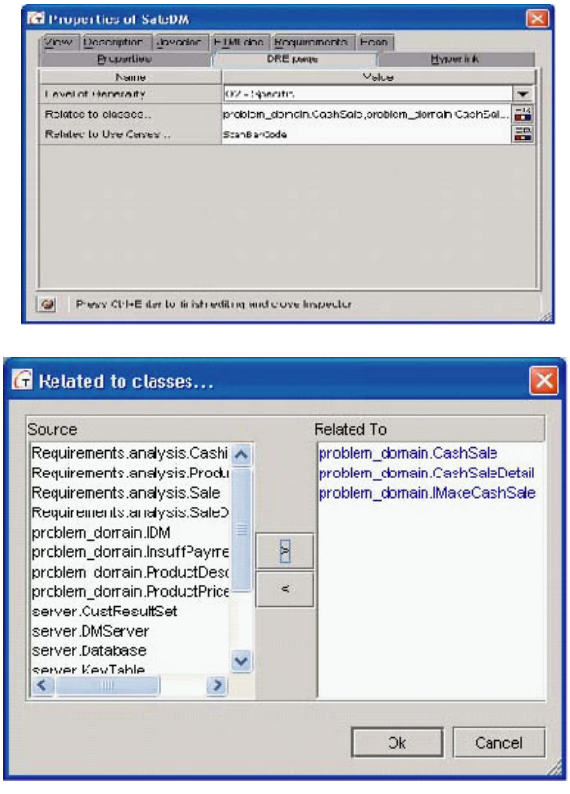


Figure 12. Relating classes to use cases

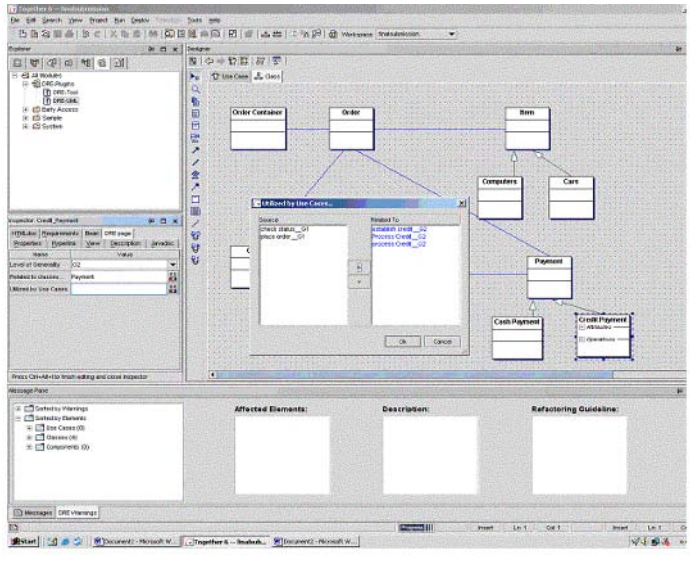


Figure 13. Violations and refactoring guidelines

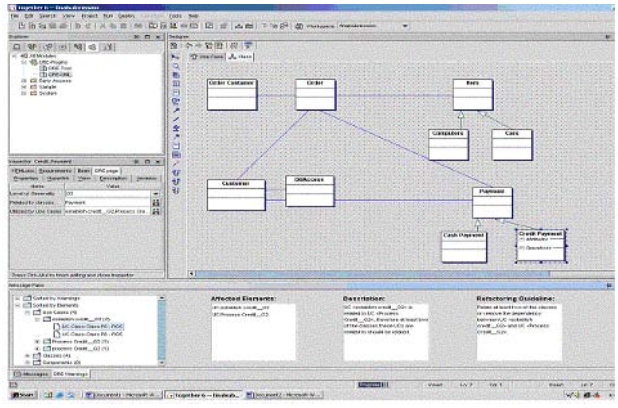
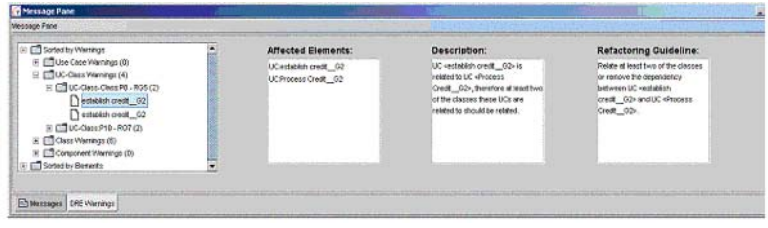


Figure 14. Reuse assessment pane



In the Reuse Definition Section, we proposed a set of formal properties and the associated refactoring guidelines for reuse definition, assessment, and analysis for use case diagrams and class diagrams. We have implemented our UML extensions into the TCC tool. Our research efforts represent a step forward in seamlessly supporting reuse in UML and Java for software designers, engineers, and developers.

In our ongoing work, we are exploring a formal model for reuse (Caballero & Demurjian, 2002) for automated reusability assessment and refactoring. We plan on expanding our UML framework and on applying our formal model concepts for automated refactoring to UML designs, including investigating the relevance of state and activity diagrams for our reusability definition, analyses, and assessment. This new framework will also be implemented in our existing prototype (UConn, 2003).

References

Basili, V.R., Rombach, H.D., Bailey, J., & Delis, A. (1990, May). *Ada reusability and measurement*. Computer Science Tech. Rep. Series, University of Maryland.

Booch, G., Jacobson, I., & Rumbaugh, J. (1999). *Unified modeling language reference manual*. Reading, MA: Addison-Wesley.

Brown, A. & Wallnau, K. (1998). The current state of CBSE. *IEEE Software,* September/ October.

Caballero R., & Demurjian, S. (2002). Towards the formalization of a reusability framework for refactoring. *Proceedings of the Seventh International Conference on Software Reuse*.

Cernuda, A., Labra, J., & Cueva, J. (2001). Verifying reuse contracts with a component model. In *6th JISBD*, November.

Chidamber, S. & Kemerer, C. (1994). A metric suite for object-oriented design. *IEEE Transactions on Software Engineer, 20*(6), 476-493.

Frakes, W. & Terry, C. (1996). Software reuse: Metrics and models. *ACM Computing Surveys, 28.*

Garlan, D., Allen, R. & Ockerbloom, J. (1995). Architectural mismatch: Why reuse is so hard. *IEEE Software*, November.

Hall, P. (1999). Architecture-driven component reuse. *Information and Software Technology, 41*(14).

Lucas, C. (1997). *Documenting reuse and evolution with reuse contracts*. Unpublished Doctoral Dissertation, Department of Computer Science, Vrije Universiteit Brussel.

McIlroy, M. (1968). Mass produced software components. *Proceedings of the NATO Software Engineering Conference*.

Meekel, J. (1997). From domain models to architecture frameworks. *Proceedings of the 1997 Symposium on Software Reusability*.

Poulin, J. (1996). *Measuring software reuse: principles, practices and economic models*. Reading, MA: Addison-Wesley.

Price, M. & Demurjian, S.A. (1997). Analyzing and measuring reusability in object-oriented designs. *Proceedings of OOPSLA'97*.

Price, M., Demurjian, S., & Needham, D. (1997). Reusability measurement framework and tool for Ada95. *Proceedings of 1997 TriAda Conference*.

Price, M., Needham, D., & Demurjian S. (2001). Producing reusable object-oriented components: a domain-and-organization-specific perspective. *Proceedings of 2001 Symposium on Software Reusability*.

Rine, D., & Nada, N. (2000). Three empirical studies of a software reuse reference model. *Software - Practice and Experience*, *30*(6).

Sarshar, M. (1996). Reuse measurement and assessment. *Proceedings of the International Workshop on Systematic Reuse*.

Schmietendorf, A. (2000). Metrics based asset assessment. *Software Engineering Notes*, *25*(4).

Software reuse executive premier (1996). Technical Report, DOD Software Reuse Initiative Program Management Office.

Steyaert, P., Lucas, C., Mens, K., & D'Hondt, T. (1996). Reuse contracts: managing the evolution of reusable assets. *Proceedings of the OOPSLA'96, ACM SIGPLAN Notices, 31*(10).

Succi, G. (1995). Reuse and reusability metrics in an object oriented paradigm. *International Journal of Applied Software Technology*, *1.*

TogetherSoft Corporation (2003). *Together control center.* Retrieved October 1, 2003, from *http://www. togethersoft.com*

Tsagias, M. & Kitchenham, B. (2000). An evaluation of the business object approach to software development. *Journal of Systems and Software*, *52*(2-3), 14.

UConn's Reuse Web site. Retrieved October 1, 2003, from *http://www.engr.conn.edu/~steve/DRE/dre.html*

Virtanen, P. (2000, April). Component reuse metrics – Assessing human effects. In K. Maxwell, R. Kusters, E. van Veenendaal, & A. Cowderoy (Eds.), *Proceedings of the combined 11th European Software Control and Metrics Conference and the 3rd SCOPE Conference on Software Product Quality*, (pp. 171-179). Munich, Germany: Shaker Publishing.

Chapter XIV

Complexity-Based Evaluation of the Evolution of XML and UML Systems

Ana Isabel Cardoso, University of Madeira, DME, Portugal

Peter Kokol, University of Maribor, FERI, Slovenia

Mitja Lenic, University of Maribor, FERI, Slovenia

Rui Gustavo Crespo, Technical University of Lisbon, DEEC, Portugal

Abstract

This chapter analyses current problems in the management of software evolution and argues the need to use the Chaos Theory to model software systems. Several correlation metrics are described, and the authors conclude the Long-Range Correlation looks to be the most promising metrics. The Long-Range Correlation measures for XML and Java files are very similar. We then identify the number of ideas that may be raised in the process of software development, and link the different behaviours of the software evolution to the Verhulst model. Finally, we analyse one industrial test case and verify that the behaviours of software evolution are represented in the Verhulst model.

Introduction

Since the 1960s, software engineers have recognised the growing costs, errors, and delays in the development and evolution of software systems are manifestations of a *software crisis* (Naur & Randell, 1969). The impact of the software crisis is huge. For example, the annual costs of software defects in the US are estimated to be up to $60 billion (NIST, 2002).

Software systems are frequently updated and, nowadays, the relative cost for maintaining the software evolution is around 90% (Seacord, 2003). Software maintenance is now an important issue, both in industry and research institutions (Grubb & Takang, 2003). The pressures for changing software systems have been identified and include (a) the technology, market, and legislation evolution, (b) the need to correct undetected errors, and (c) the increase of understanding of the system by the users after the release installation (Lehman & Belady, 1985). As result of the pressures for system change, either the software system adapts to these changes or becomes less useful until, ultimately, it must be discarded. After analysing multiple versions of the IBM OS/360 operating system, Lehman has suggested that the evolution of a software system is subject to three informal laws: (a) *continuing change*, that states a program used in a real-world environment must change, (b) *increasing entropy*, that prescribes the program structure becomes more complex unless efforts are made to avoid the complexity, and (c) *statistically smooth growth*, that enounces the global system metrics appear locally stochastic in time and space but are self-regulating and statistically smooth (Lehman, 1980).

Nowadays, software systems have a wide range of applications, such as reactive systems for Machine Control, symbol processing in Artificial Intelligence, and number crunching for Simulation. Developers now have available many design and programming languages, each one oriented for a specific range of applications. The object-oriented design and programming languages have been adopted for the development of many applications because they are closer to the real world, and data are not shared, which reduces overall system coupling as there is no possibility of unexpected modifications to shared information. The Unified Modeling Language (UML) (Fowler & Kendall, 1999) and Java (Arnold & Gosling, 1997) are two examples of widely used object-oriented design and programming languages.

Project managers recognise that the higher the semantic organisation of the computer applications is, the easier the software understanding and the lower the cost of system evolution. Informally, the semantic organisation of a program is given by the regularity presence of software elements. Object-oriented systems enforce regularity in some of the software elements, such as data encapsulation and class hierarchies. Our research focuses on the regularity of the software elements in UML diagrams and Java programs that vary according to the programmers and to the project goals. For example, programmers with different practices create, in the UML sequence diagrams, different sequences of message interactions between the same classes for the same software system.

Many project teams and researchers have proposed metrics to quantify and control the software development and evolution (Fenton & Pfleeger, 1997). The results, however, fell below the initial expectations. One possible reason for this failure is that the current

metrics focus only on some aspects of the software development and their products. In this chapter, we use another approach, based on *complex* or chaotic systems, to assess the overall organisation and evolution of the software systems.

The purpose of this chapter is twofold: (a) to show how to measure the semantic organisation of object-oriented systems; and (b) to identify the behaviours that the evolution of software systems may depict. These two goals help us to validate the laws of software evolution and to control the adaptation of software systems.

Software as a Complex System

Complex, or chaotic, systems have been adopted in the representation of many areas of activity, such as biology, economics, and physics. Complex systems share certain features like having a large number of elements, being sensibility dependent on the initial conditions and representing an extended space for evolution (Gell-Mann, 1995). Also, the behaviour of complex systems cannot be understood from the sum of the behaviours of its parts.

Likewise complex systems, software systems are made of many software artefacts, such as design UML diagrams, specifications expressed in Object Constraint Language (OCL) (Warmer & Kleppe, 2003), source programs in Java, object files, and test cases. The attributes and the behaviour of the individual software artefacts are different from the system attributes and behaviour. Also, systems are constantly updated and the evolution space is very large. Therefore, software systems may be modeled as complex systems.

Whatever representation scheme is selected, each software artefact is a string, or a sequence of code symbols (CS). UML diagrams and test cases may be stored in eXtended Markup Language (XML) (Ray, 2001). The source program symbols are those of the programming language, Java, for example, and the object file symbols are the binary digits. For the high-level languages, we are interested on the reserved words that have a meaningful relationship with the user goals and provide actions. We do not care about the regularity imposed by the "sugar syntax" of the UML and the XML languages, neither to the attributes and constants.

The symbols are organised in two levels, syntactical and semantic. The syntactical organisation is a well-known field of study (Appel, 2002). There are many ways to describe the semantics of programming languages, for example, attribute grammars (Knuth 1971) and denotational semantics (Winskel, 1993). The regularity of the semantic language components, however, is still largely unknown. We advocate that it is possible to quantify the level of the semantic organisation without the explicit determination of the program semantics.

Software Process Metrics

The impact of metrics in the evaluation of software processes has been studied. The analysis may be focused on specific areas, such as the measurement theory (Briand, 1995) and the project team classification (Paulk, 1993). In this chapter, we divide the analysis of the software process metrics according to the underlying system process modeling, classical and Chaos Theory based.

Classical Metrics

Classical modeling observes software processes as a coordinated set of activities, each one manipulating different software products. The metrics based on classical modeling of systems measure specific attributes of the system. The global understanding of the software development process and products is expected to be some sort of attribute summation. We divide the classical metrics in two categories, plain and entropy metrics.

Plain Metrics

Plain metrics are concerned with the overall value of the attributes. Many software product metrics are available, such as dimension, cohesion, and coupling. Classical metrics, such as the number of lines of code (LOC), are intuitive and have been successfully used in important aspects of software projects, such as cost estimation (Boehm, 1981). Yet, the classical software product metrics have provided limited use in the measurement of software processes (Kitchenham, 1998). Reasons for such failure include (Cardoso, 2001)

- Product current metrics depend on the selected language, thus making it impossible to compare systems implemented in different languages.
- Software products are modeled by different representation schemes, such as informal and graphic representation in the requirements phase and programming languages in the code phase. Therefore, classical metrics only measure particular aspects of the software processes.
- Current software process metrics measure some attributes of the process. Because our knowledge is limited, we are unable to quantify global values of the software process.

Entropy Metrics

Entropy metrics have been used to study the regularity of the distribution of symbol frequencies. Consider a string $s \varepsilon S^+$, with each $w_i \varepsilon S$ having the p_i frequency. The equations (1) and (2) define, respectively, the Shannon and the Rény entropies.

$$H = -\sum_i p_i \log p_i \quad (1)$$

$$H_r = \frac{1}{(1-\alpha)} \sum_i p_i^{\alpha} \quad (2)$$

In the Shannon entropy, the greater and the lesser frequencies contribute equally to the result. In the Rény entropy, the greater frequencies are overcome if $\alpha>1$. If $0<\alpha<1$, the lesser frequencies are overcome.

We identified the entropy metrics for valid programs, with a purpose, and randomly generated programs with syntactic correctness. We verified that there is no difference between the two sorts of programs. Therefore, we directed our attention to the correlation metrics.

Correlation Metrics

Complex modeling realises that merging different activities results in the formulation of new behaviours.

The correlation metrics are often used in measuring complex systems. Correlation metrics quantify the connection between the same symbol elements in different string positions. The short-range correlation metrics, such as the Markov chains, relates elements closely positioned, and the long-range correlation metrics (LRC) filters short-range fluctuations.

Brownian Cumulative Walk

LRC has been used to identify the basic structure in different areas, such as DNA (Peng, 1992) and human writings (Schenkel, Zhang, & Zhang, 1993). In this chapter, we adopt the Brownian cumulative random walk (RW) (Peng, 1992) to measure the LRC.

The symbols are translated with a balanced numeric code, that is, the sum of all codes is equal to zero. The RW is identified as

$$RW_0 = 0$$

$$RW_i = RW_{i-1} + Cod(S_i)$$

From the RW, we identify the characteristic function, F(l), as the root of the mean square fluctuation about the average of the displacement l. l is the distance between 2 points of RW in the horisontal axis, and y is the distance between 2 points of RW in the vertical axis. $Dy(l,l_0)$ is equal to $y(l+l_0)$-$y(l_0)$.

$$F^2(l) = \overline{[\Delta y(l, l_0)]^2} - \left[\overline{\Delta(l, l_0)}\right]^2 \quad (3)$$

Figure 1 depicts one example of the F(l) characteristic function. The F(l) characteristic function can distinguish two possible types of behaviour:

1. If the string is uncorrelated, or there are local correlations extending up to the characteristic range (e.g., Markov chains and sequences generated by regular grammars), then $F(l) \gg l^{0.5}$.
2. If the correlations are "infinite," then $F(l) \approx l^{\alpha}$ and $\alpha \neq 0.5$.

The α value is extracted from slope of the F(l) characteristic function, removing the lowest and the highest values. The lowest initial values are considered to be noise and, therefore, are useless for alpha estimation. The highest l values must be discarded because the index difference is close to the sample size.

The α values for valid programs with a purpose, and for randomly generated programs with syntactic correctness, are different (Cardoso, Crespo, & Kokol, 2004). Well-organised programs hold a values furthest away from 0.5, around 0.7 and 0.8.

We developed the *Complexity Analyser* tool, running on Windows, to measure the a values in several languages such as C, Java, UML, and binary files. More information may be found at http://lsd.uni-mb.si/eng/research/software/SCA.

Restrictions on the α Determination

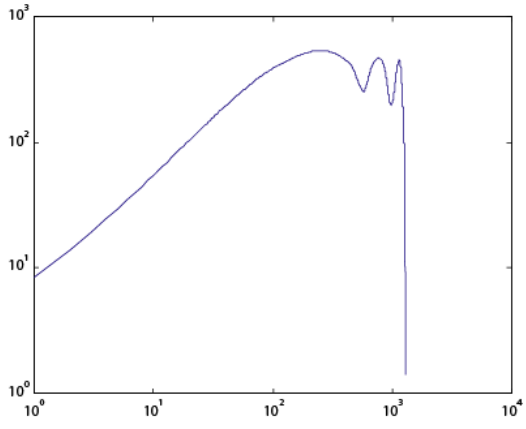
The computational cost of the a determination is $O(N^2)$, with N equal to the string size. We consider the a value to be valid if, in the best fitting curve obtained by the least squares method, the slope has an adjustment coefficient greater than 0.7 (McClave, 1991). We conducted some experiments on C language programs, where the CS size is equal to 32, and the a values were valid only for files containing more than 700 reserved words. We also have verified a strong correlation between the a values in source and in the derived object files (Cardoso et al., 2004). Hence, project managers may select the best representation to overcome the restrictions on the computational costs and on the string lengths.

For object files, the code symbols are the binary digits and CS is equal to 2. Therefore, the major restriction is the computational cost of the value.

For Java programs, the symbols are reserved words of primitive types (e.g., *int* and *boolean*), qualifiers (e.g., *unsigned* and *public*), class declarations (e.g., *class* and *implements*), and statements (e.g., *while* and *throw*). CS is equal to 47 and, thus, the sample dimension becomes a major restriction for Java programs.

UML version 2.0 defines 13 types of diagrams. We intend to study the correlation between the organisational levels of the UML diagrams and the object code files. Therefore, we restrict our analysis to the data and to the behaviour UML diagrams.

Figure 1. The shape of the F characteristic function



UML provides diagrams for data modeling, such as Class (CD), Object and Package diagrams. We selected CDs because they describe the basic structure of the system. The reserved words of the CDs are the Java primitive types, qualifiers and class statements. For the class diagrams, the CS value is equal to 22.

UML provides diagrams for behavioural modeling, such as Activity (AD), Sequence, and Use Case diagrams. We selected ADs, because they model the control flow between the operations on the classes in the system. The reserved words for the ADs are activities, branching (*branch* and *merge*), object flow, and synchronisation (*fork* and *join*). For the activity diagrams, the CS value is equal to 6.

Reverse Bifurcations in Software Development

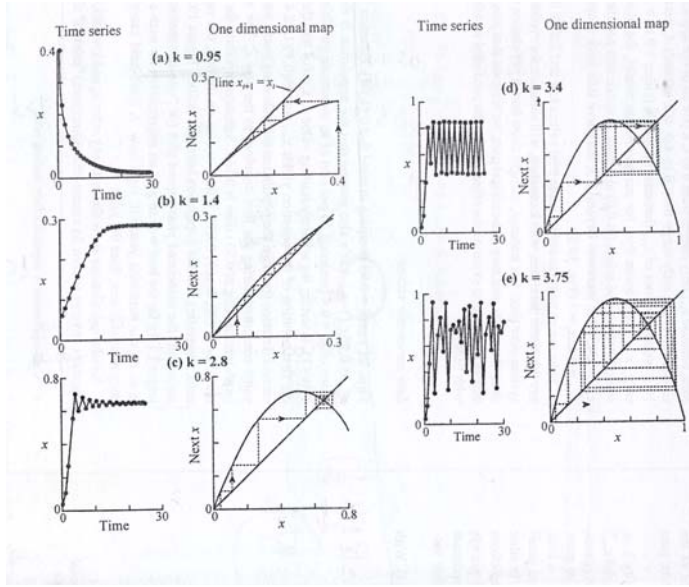
The evolution of organisations and populations may be modeled by the logistic map, or Verhulst model (Devaney, 1989). We advocate that the logistic map also models the evolution of the computer applications.

Logistic Map

The sequence of elements $X_t \varepsilon [0,1]$ is derived by the function depicted in (4)

$$X_0 > 0$$
$$X_{t+1} = kX_t(1 - X_t) \quad (4)$$

Figure 2. Behaviour of the sequence



The terms kX_t and $-kX_t^2$ may be seen, respectively, as the balance of natural "births" minus natural "deaths," and the number of extra "deaths" due to the population overcrowding. The function depicted in (4) considers that the population grows proportionally to the current population, if lower than 1, and declines with a slope equal to the distance to 1, if greater than 1. The behaviour of the X_t sequence depends on the k coefficient, known as carrying capacity, and is depicted in Figure 2.

- If k is lower than 3.0, X_t tends to a fixed value. Figure 2 depicts, on the left, the cases for k equal to 0.95, 1.4, and 2.8.
- If k is greater than 3.0 and lower then 3.569945…, X_i oscillates between a fixed number of values equal to a power of 2 (2 values if k is lower than 3.44949, 4 values if k is lower than 3.54409,…). Figure 2 depicts, on the upper right, the case for k equal to 3.4.
- If k is greater than 3.569945…, X_i changes chaotically. Figure 2 depicts, on the lower right, the case for k equal to 3.75.

The Iteration Process

We advocate the behaviour of the iteration process is linked to the number of ideas generated in the generic process of software development. The number of ideas, about the computer application and its implementation, are depicted in the upper part of Figure 3.

In the early stages of the software process life cycle, the number of ideas is very large. Some ideas represent possible valid solutions; others lead to invalid or impossible implementations. In the early stage, creativity is large, entropy level is high, and information content is low. The semantic organisation is low, and the α values are close to 0.5.

In the intermediate stages of the software process life cycle, the project passes through a convergence of ideas, where invalid and incorrect choices are discarded. In the intermediate stages, a number of ideas may be further explored, in the search for a locally optimal solution. The intermediate stages conclude with the selection of one single idea, probably a very complex one.

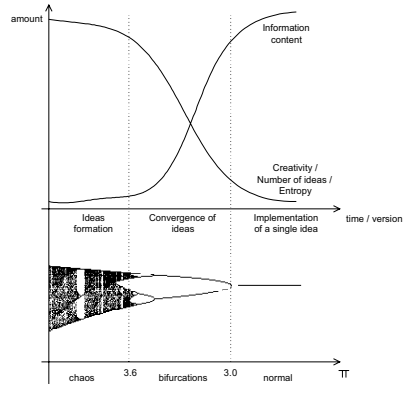
In the last stages of the software process life cycle, the project members work with one single idea, the creativity and the entropy reach a minimum level, and the information content is high. The semantic organisation is high, and the α values are much higher than 0.5.

Each cycle of the iteration process is an instance of the generic software process, where the previous versions have a strong influence on the number of ideas in the new cycle. Some version implementations are simple adaptations, with one unique idea, and the early and intermediate stages of the software development are absent. Other version implementations represent major changes, many ideas are explored, and the early and intermediate stages of the software development play a major role.

To quantify the behaviour of the version sequence, using the semantic organisation, we first normalise the α values to

$$\alpha_{nor} = 2 * |\alpha - 0.5| \tag{5}$$

Figure 3. The software development process and the logistic map



And *Equation 4* is rewritten to

$$k = \frac{\alpha_{nor(t+1)}}{\alpha_{nor(t)}\left(1-\alpha_{nor(t)}\right)} \tag{6}$$

Test Cases

We explored our ideas with Java applications registered in the JBOSS repository. JBOSS is a J2EE-based application server, available at http://www.sourceforge.net/projects/jboss. The repository contains more than 600 programs, each one with a number of versions. We directed our attention to programs that suffered adaptations more than 200 times.

Invariance on the Semantic Organisation

To verify if the a values remain constant along the different software artefacts in the same process life cycle, we evaluated the semantic organisation values of the UML diagrams and the binary code files generated from the UML diagrams. We did not determine the α values for the Java source files, because the required size for the number of reserved words is too high and each source file contains less than 400 lines of code (LOC).

The class and activity diagrams are not available at the repository, therefore we reversed engineered the diagrams from the Java source files. In the reverse engineering of CDs, we simply collect the class and attribute definitions. In the reverse engineering of ADs, we adopted the rules of (a) an activity represents a sequence of assignment statements

Figure 4. Correlation between UML and Java α values

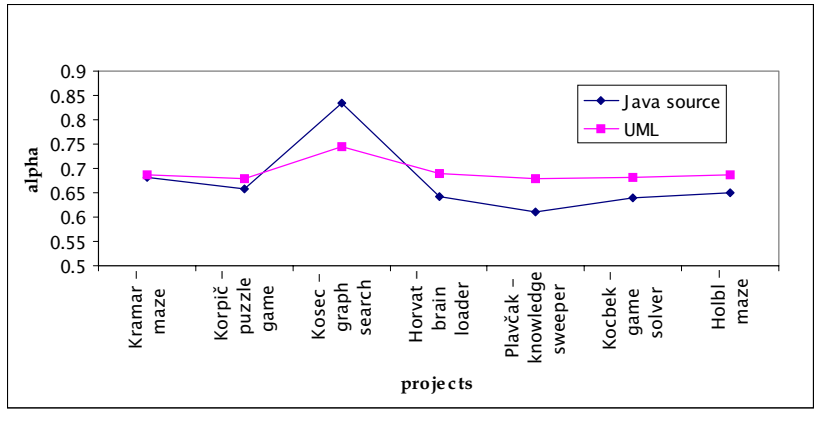
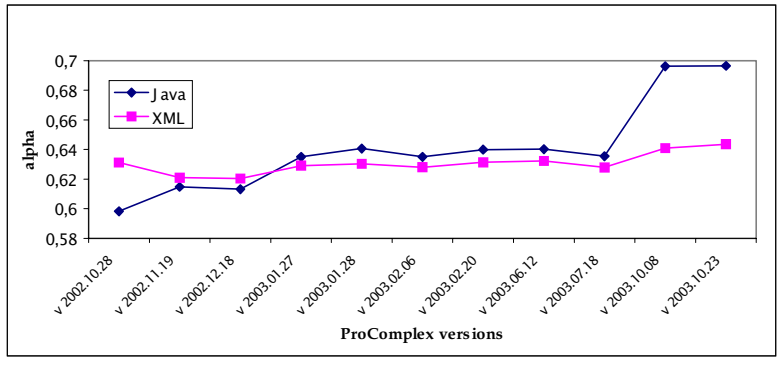


Figure 5. Correlation between XML and Java α values



or function calls, (b) an object flow represents a method evocation, a method return. or an exception *throw*, and (c) the branch and merge represent the outer limits of the Java control-flow statements of *do*, *for*, *if*, *switch,* and *while*.

To make the results statistically valid, we identify the a values for a pair of files and remove in the α calculation the seldom-used reserved words of *short*, *final*, *transient* and *volatile*.

Figure 4 depicts the a values, for seven different packages, measured in the UML and in the Java files. The results suggest there is a strong correlation between the UML and the Java α values.

Figure 5 depict the α values, for 11 versions of the ProComplex package in the XML and in Java files. This package measures and plots α and k values in different languages, from object code to Java and C (Kernighan & Ritchie, 1988) source files. Again, the results suggest a strong correlation between the XML and the Java α values.

Figure 6. k evolution at the BeanMetaData versions

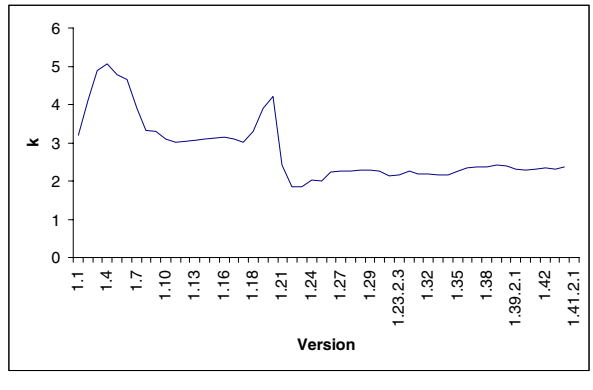
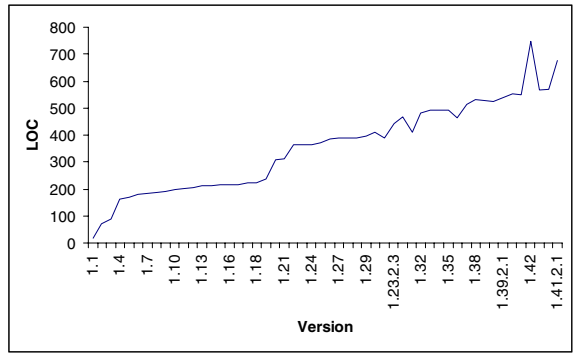


Figure 7. LOC evolution at the BeanMetaData versions



Behaviour of Class Versions

The behaviours, depicted in Figure 2, are observed in the JBOSS packages. In this section, all Π values are collected in the Java bytecode files.

Figures 6 and 7 depict, respectively, the k values and the LOC for the BeanMetaData.java versions. The first version was stored in the repository on July 2000, and we conducted the analysis until version number 52 was stored in the repository on October 2002.

The chaotic behaviour, as a result of idea formation, is depicted from version 1.2 to 1.7, and between versions 1.19 and 1.20. The behaviour reveals convergence of ideas in versions 1.1 and versions between 1.8 and 1.18. From versions 1.21, the behaviour remains stable.

Despite the change of behaviours in the BeanMetaData.java versions, from chaos to stability, the number of source lines of code grows steadily from 17 to 700. The independence between the program behaviour and the number of source lines of code also is verified in many other Java packages.

We also informally compared the behaviour of several JBOSS packages with the revision comments. In the JBQLCompiler, a parser, the a values reveal the initial behaviour to be chaotic, becoming stable after the application of the parsing theory. This result suggests the complexity-based method may be used to support project managers in the control of software evolution.

Conclusion

The complex system model provided an excellent framework in the analysis of the evolution in UML and Java system versions. With the LRC metrics, we are able to quantify the semantic organisation of the programs and how it evolves. Moreover, the LRC α metric reveals that, the values are strongly correlated between the different products

generated in a software process life cycle. These results must be checked for other design and programming paradigms, for example, to compare highly structured and low structured programming languages.

The evaluation of the α values also reveals some limitations in the file size. The file size must be sufficiently high to make the results statistically valid, but not too high due to the computational costs of the LRC evaluation. It is highly desirable to identify a good estimation for the minimal string size, in terms of the alphabetic dimension CS. Also, research must be conducted to identify other LRC metrics that identify the semantic organisation and strongly correlate the values on different software artefacts of the same project version, and impose less limitations in the artefact size and in the computational costs. While a more precise identification of the string limits remains unavailable, the strong correlation between the α values of different products gives some freedom for managers to select the project samples in the project.

References

Appel, A. (2002). *Modern compiler implementation in Java* (2nd ed.). New York: Cambridge University Press.

Arnold, K. & Gosling, J. (1997). *The Java programming language* (2nd ed.). Boston: Addison-Wesley.

Boehm, B. (1981). *Software engineering economics*. Englewood Cliffs, NJ: Prentice-Hall.

Briand, L. & Eman, K.E. & Morasca, S. (1995). *On the application of measurement theory in software engineering.* Technical Report International Software Engineering Research Network #ISERN-95-04.

Cardoso, A.I., Crespo, R.G., & Kokol, P. (2001). An alternative way to measure software — A measure from complex system theory. *Proceedings of World Multiconference on System Cybernatics and Informatics,* (pp. 213-216). Orlando, FL: IEEE Computer Society.

Cardoso, A.I., Crespo, R.G. & Kokol, P. (2004). Complexity-based metrics for the evaluation of the program organization. In R. Dobrescu & C. Vasilescu (Eds.), *Interdisciplinary approaches in fractal analysis*. Bucharest, Romania: Publishing House of the Romanian Academy.

Devaney, R. (1989). *An introduction to chaotic dynamical systems*. Boston: Addison-Wesley.

Fenton, N.E. & Pfleeger, S.L. (1997). *Software metrics, a rigorous and practical approach* (2nd ed.). London: Thomson Computer Press.

Fowler, M. & Kendall, S. (1997). *UML distilled: A brief guide to the standard object modelling language* (2nd ed.). Boston: Addison-Wesley.

Gell-Mann, M. (1995). What is complexity? *Complexity, 1*(1), 16-19.

Grubb, P. & Takang, A. (2003). *The software maintenance, concepts and practice* (2nd ed.). London: World Scientific.

Kernighan, B.W. & Ritchie, D.M. (1988). *The C programming language* (2nd ed.). Englewood Cliffs, NJ: Prentice-Hall.

Kitchenham, B. (1998). The certainty of uncertainty. *Proceedings of the European Software Measurement Conference* (pp. 17-25). Antwerp, Belgium: Technologisch Instituut KVIV.

Knuth, D.E. (1971). Semantics of context-free languages. *Mathematical Systems Theory, 2*(2), 127-145.

Lehman, M.M. (1980). Programs, life cycles and laws of software evolution. *IEEE Special Issue on Software Engineering, 68*(9), 1060-1076.

Lehman, M.M. & Belady, L.A. (1985). *Program evolution: Processes of software change.* San Diego, CA: Academic Press.

MacClave, J., Benson, P.G. & Sincich, T. (1991). *Statistics for business and economics* (7th ed.). Englewood Cliffs, NJ: Prentice-Hall.

Naur, P. & Randell, B. (1969). *Software engineering.* Report on Conference sponsored by NATO Scientific Affairs Division, Garmisch-Paterkirchen.

NIST (2002). The economic impact of inadequate infrastructure for software testing. *Planning Report 02-03 for National Institute for Standards & Technology*, Gaithersburg, MD.

Paulk, M.C., Curtis, B., Chrissis, M.B., & Weber, C.V. (1993). Capability maturity model for software. *IEEE Software, 4*(10), 18-27.

Peng, C.K., Buidyrev, S.V., Goldberger, A.L., Havlin, S., Sciortino, F., Simons, M., & Stanley, H.E. (1992). Long-range correlations in nucleotide sequences. *Nature, 356,* 168-170.

Ray, E.T. (2001). *Learning XML*. Sebastopol, CA: O'Reilly and Associates.

Seacord, R., Plakosh, D., & Lewis, G.A. (2003). *Modernizing legacy systems: Software technologies, engineering processes and business practices.* Boston: Addison-Wesley

Schenkel, A., Zhang, J., & Zhang, Y. (1993). Long-range correlations in human writings. *Fractals, 1*(1), 47-55.

Warmer, J. & Kleppe, A. (2003). *The object constraint language* (2nd ed.). Boston: Addison-Wesley.

Winskel, G. (1993). *The formal semantics of programming languages*. Cambridge, MA: MIT Press.

Chapter XV

Variability Expression within the Context of UML: Issues and Comparisons

Patrick Tessier, CEA/List Saclay, France

Sébastien Gérard, CEA/List Saclay, France

François Terrier, CEA/List Saclay, France

Jean-Marc Geib, Université des Sciences et Technologies de Lille, France

Abstract

Time-to-market is one of the most severe constraints imposed on today's software engineers. The increasing complexity of systems has also shortened the time available for designing them. Several solutions have therefore been proposed to decrease the time and cost of producing applications. This chapter presents the product line paradigm as an effective solution for managing both the variability of products and their evolutions. The product line approach calls for designing a generic and parameterized model that specifies a family of products. It is then possible to instantiate a member of that family by specializing the "parent" model or "framework." In describing the latter, designers need to explicitly model variability and commonality points among applications. The following discussion explains in detail how UML models express

these different requirements. We then describe specific extensions of UML profiles and the way they are used in various product line methodologies.

Introduction

Thanks to increasing storage and processing capabilities, software-based applications now provide more and more functionalities. This is achieved at the cost of ever greater complexity and heavier developer workloads. Market conditions have fostered sharp competition that has accelerated the software production cycle, while maintaining the same high quality development criteria.

To reduce time and cost, companies must therefore be able to capitalize on their work. Their systems must be maintainable throughout their service life. They also must be updatable to integrate future evolutions (new functionalities, new hardware, improved optimization, etc.). This evolutivity is one of the most important features of industrial systems at a time when service life requirements are being stretched to a maximum. For example, automotive systems are adapted to each type of automobile and may need to evolve with changes in automobile design or the advent of new hardware platforms that are cheaper and more efficient.

One approach to managing the evolution of software-based systems consists of designing each new application as an application family (Bosch, 2001). Any subsequent evolution in that application is then viewed as a new member of the family.

The application family paradigm was first proposed by (McIlroy, 1968; Parnas, 1979). The main focus behind their idea was to foster reuse of the models for a whole set of applications. An application is then seen as a specific instance of an application family called an application domain. This is known as the "product line" approach.

Clements later defined the software product line as follows:

"A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy 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." (Clements & Northrop, 2001)

From this definition, one can deduce that a software product line relies on its capacity to use and compose common and specific features of an application domain called a framework. Each specific member of an application domain then results from one specialization of a framework.

The main requirement for modeling a product line is the ability to depict the commonalities and variabilities of an application domain in the context of a framework. Several formalisms such as FAST (Harsu, 2002), FODA (Kang, Cohen, Hess, Novak, & Peterson, 1990), and GenVoca (Batory, Chen, Robertson, & Wang, 2000) are still available to address this need.

All of these product-line-based approaches rely on their own formalisms to model the variability features of applications.

The Unified Modeling Language (UML) (OMG, 2002), which was developed to deal with certain aspects of software engineering (architecture and behavior specifications for distributed object systems), has now become the lingua franca for software-based applications (Franch & Ribo, 1999). In keeping with this trend, some product-line-based approaches have been ported to UML to make the most of its now broad acceptance. For all these reasons, our purpose here is to provide a survey of several product line approaches. This chapter focuses on two points: firstly, UML2 native facilities for modeling product line frameworks; secondly, UML-based (Catalysis, Kobra, UML-F, CAFÉ) approaches. Special attention is therefore devoted to extensions which improve the usability of UML2 for variability modeling management. The following table gives a quick overview of the various approaches that are examined in greater detail in the rest of the chapter.

Case Study

Each of the concepts described in this chapter is illustrated by a simple case study. The example used is a robot driving system. This example is taken from a project baptized JOSEFIL (http://www.ensieta.fr/mda/JOSEFIL).

The common requirement for the JOSEFIL project is to drive a robot to various predetermined positions in a room. The robot is equipped with sensors that ensure both current position acquisition and possible obstacle detection.

There are then three variable requirements corresponding to different robot driving modes:

Table 1. Summary of the features provided by various product line approaches

	Variability in a structural model	Variability in an interaction model	Variability in behavioral model	Decision Model	Support Tool
Catalysis	Pattern template + Component	Nothing specific	Extension of UML use case diagrams	Extension of UML use case diagrams	SmartDraw seems to support this approach
CAFÉ	Apply <<optional>> stereotype to entity with OCL constraints	UML2.0 combined fragment of sequence diagrams	Apply <<variant>> and <<optional>> stereotypes to actor and use case	Nothing specific	No
UML-F	Pattern template	Stereotyped action in sequence diagrams to model repetition or optional features of message	Nothing specific	Instantiation case depicted via textual scenarios	No
Kobra	Stereotype <<variant>> on components and methods	Apply <<variant>> stereotype to message of sequence and collaboration diagrams	Apply <<variant>> stereotype to use case transitions of activity diagrams and state/transitions of state-machine diagrams	Parameterized checklist	A prototype exists

- **Automatic mode** – The system itself calculates the paths to each specified position and also is able to avoid automatically detected obstacles. This mode calls for parameterizing path calculation to ensure that the robot reaches each of the specified positions in the fastest, most cost-effective way.
- **Partially assisted mode** – In this mode, the system still calculates the robot paths automatically, but, on detecting an obstacle, requires interaction with a user to manually bypass it.
- **Manual mode** – The user drives the robot via a remote interface.

Based on the requirements, it is possible to define a product family of JOSEFIL robot control systems, some of which offer only manual control, others manual plus automatic control, and so forth.

Variability Modeling in UML

As shown, modeling a family of products entails building a framework made up of common and variable elements. The following section describes the native concepts provided by UML in its standard form for modeling such a framework. The section is organized around three subsections covering three model types—structural, behavior, and interaction.

Structural Model

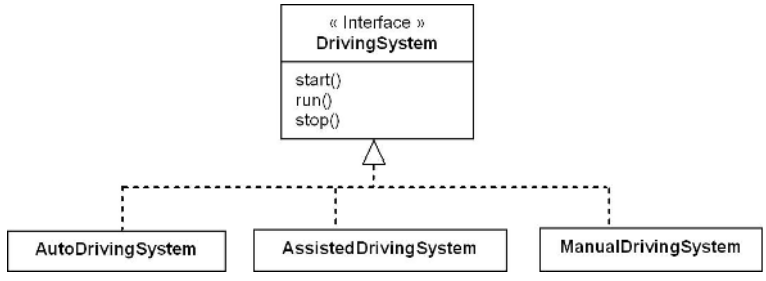
The structural model is an entity/relation-based model consisting of basic elements (e.g., classes, components) and their relationships (e.g., associations, dependencies). The rest of this section is dedicated to describing UML structural concepts that afford variability modeling support.

Interface

An interface is a declaration of a set of features that may be structural (e.g., attributes, association) and/or behavioral (e.g., operation, reception). This concept is generally used to model a kind of contract that has to be realized by at least one classifier (e.g., class, use case) in the application. In this case, the classifier realizing an interface is associated with it via an implementation relationship.

The interface concept also affords the possibility of expressing variability of realization in one model. When considering a system as a black box, it is possible to declare a set of provided features by defining its interfaces. Later, when considering it as a white box (i.e., detailing the inside of the box), it may be possible to model different ways of realization, for example by introducing classes that realize the different interfaces.

Figure 1. Three variations afforded by an interface for three possible driving modes



Various realizations of a same interface can then be proposed as a function of constraints or conditions defined by the user.

The interface described in Figure 1 declares that the driving system must provide three main services: start, stop, and run. The diagram also requires that, for each different driving mode, these services be realized by the associated classifiers (in this case *classes*).

If the requirements subsequently evolve (e.g., by specifying a new way to drive the system), it is then only necessary to add a new realization link with the provided interface.

Finally, it is possible to restrict the implementation relationship by adding to the model constraint declarations that specify conditions for choosing a particular realization from among the various possibilities. For this purpose, UML proposes to write OCL[1] constraints (OMG, 2003). In the example depicted in Figure 1, it is possible to add the constraint *{xor}* between all classes that implement the interface. This means that these implementations cannot coexist in the same specific application.

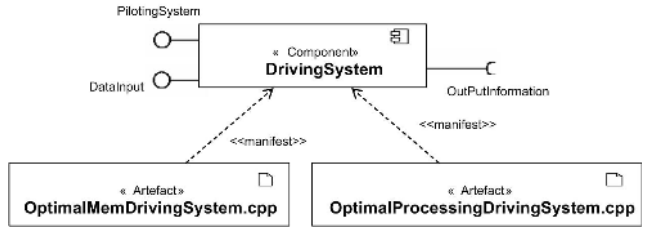
Component

A component is a modular piece of a system that encapsulates its contents. It is defined in terms of provided and required interfaces. As such, a component serves as a type whose conformance is defined by these interfaces. A component implements provided interfaces and uses operations defined in required interface. Larger "chunks" of a system functionality may be assembled by reusing components as parts in an encompassing component or assembly of components, and wiring them together through their required and provided interfaces.

Product line approaches are often associated with the component concept (Christian Bunse, 2001; D'Souza & Will, 1998). The first reference to a product line was in fact associated with the component concept (McIlroy, 1968). Components are used to insert variability into models of product families.

In UML 2.0, a component is first manifested as one or more artifacts that are the specification of a physical piece of information used or produced by software. This information can be a source or binary file. Variability is expressed by choosing one of several artifacts. In this context, the component is viewed as a black box.

Figure 2. Two possible code artifacts for the DrivingSystem component



The driving system has two variants and therefore has two artifacts. These artifacts are binary files optimized for different hardware architectures (see Figure 2). They are linked to a *DrivingSystem* component model by a "*manifest"* relationship indicating that the component is implemented by a tangible physical entity. The *OptimalProcessingDrivingSystem* artifact is optimized for processing capacity; and another artifact, *OptimalMemDrivingSystem*, is optimized for memory capacity.

Variability then also allows several realizations of a component. A component can have several models that describe its architecture and behavior. In this context, it is viewed as a white box. In designing a component, engineers then choose one from among several possible realizations.

In Figure 3, the *DrivingSystem* component has two realizations. Different versions of the driving system have been produced. This component proposes two modeling variations: *DrivingSystemV1* and *DrivingSystemV2*.

Template

In UML, templates are one of the most straightforward ways of expressing variability by parameterizing a model element. In general, any element can be assigned a parameter and thus become a template. In practice, template elements usually include Class, Package, and Operation.

Variability is achieved here by parameter declarations attached to model elements. To specialize a framework element, it then suffices to give specific values to the parameters of that element. This is known as binding a template, and the result is a bound element.

Figure 4 illustrates the use of a package template. Our system is required to send different

Figure 3. Two variations of component model design

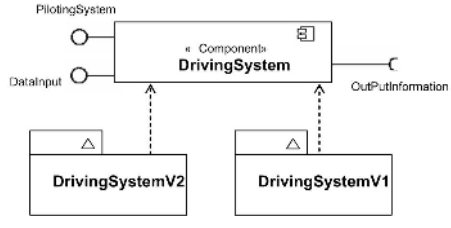
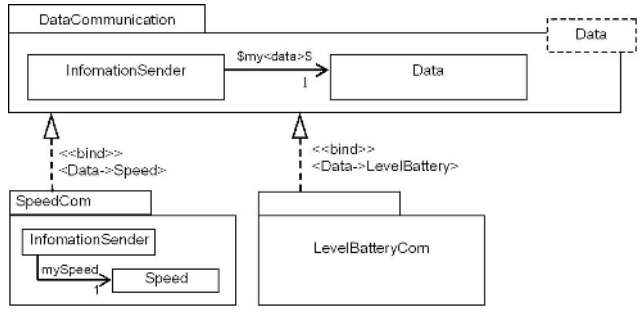


Figure 4. Use of a template package for data communication management to construct two variations: SpeedCom and LevelBatteryCom



kinds of information to a remote user display. In our framework, we would like to ensure that the type of information sent by the system can be modified. To achieve this goal, we introduce a package template in which the type of information is a parameter. It is then possible to bind the parameter with different types, for example the *BatteryLevel* type. This mechanism can also be applied to managing other information such as *Speed or Position*.

Figure 5 depicts use of a class template. The left-hand side of Figure 5 shows the specified generic sensor class. This class has two parameters: *DataType* specifies the type of the data the sensor is managing; and *freshness* specifies the time during which the acquired value is valid after its *timeStamp* (acquisition date). The right-hand side of Figure 5 shows a bound class that models a speed sensor class.

Framework

A framework is a stereotyped package («framework») that usually consists of classes, patterns, or templates. It contains elements of a reusable architecture for all members of the system family. A framework can be specialized, using an inheritance relationship, to yield a specific application.

Thanks to this entity, reuse elements of a product family can be grouped into several framework packages. A system family project can thus be structured into several

Figure 5. Expressing variability by binding a class template for a generic sensor

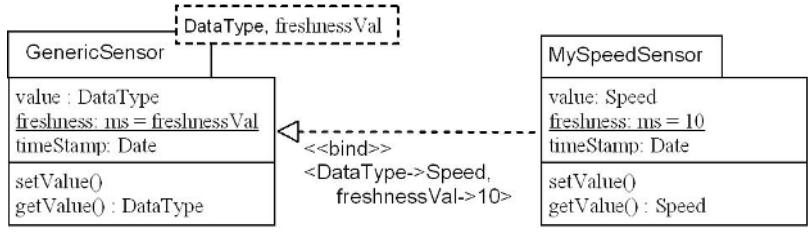
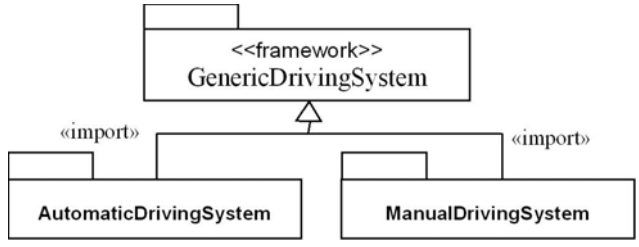


Figure 6. Framework package of driving system with two specializations



framework packages and each package specialized to build a specific package. Each member of the family is then a combination of specific packages. This allows designers to organize a system family model by expressing large scale variability.

In Figure 6, reused elements of *GenericSystemDriving* are grouped into a framework package. To obtain an automatic or manual driving system, all elements of the *GenericSystemDriving* package are copied and adapted (e.g., by binding template elements) into two new packages: *AutomaticSystemDriving* or *ManualSystemDriving*.

Generalization

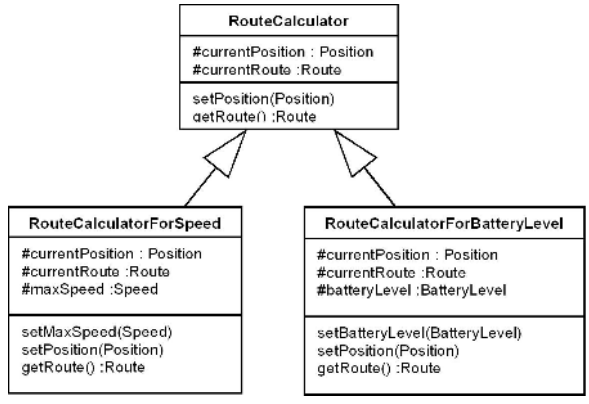
A generalization is an inheritance relationship connecting two categories of entities, generally parent elements and child elements. A parent may have various children, but a child also may have several parents (in some cases more than two!). Parent entities are general classifiers and child entities more specific classifiers. Each specific classifier inherits the features of its parent classifier.

This is used when variation points are set by methods that need to be implemented for every application or when an application needs to extend a type with additional functionalities (Svahnberg & Bosch, 2000). Such a mechanism allows factorizing of commonalities between several applications. Commonalities are contained in more generalized classes. Variability is contained in each class that has an inheritance link with the more generalized class.

Note that a parent class can be abstract. This kind of class cannot be directly instantiated and has to be generalized by a concrete class.

The *RouteCalculator* class can have two variations (see Figure 7). In the first case, the method calculates a route to make the robot faster. In the second case, the method calculates a route to minimize energy consumption. Both variation points are introduced via the following classes: *RouteCalculatorForSpeed* and *RouteCalculatorForBatteryLevel*. These cases present the following commonalities: the *setPosition* method and the attributes *currentRoute* and *currentPosition*.

Figure 7. Inheritance mechanism for calculating robot route: Two possible calculations



GeneralizationSet

GeneralizationSet defines a particular set of generalization relationships that describe the way in which a specific Classifier may be partitioned. This element indicates whether the sets of specialized classifiers share instances. It can inform the user that no instances of the general classifier are instances of specialized classifiers. This element also indicates whether there is intersection between sets of instances of specialized interfaces.

In Figure 8, the *RouteCalculator* class is specialized by four subclasses: *RouteCalculatorForSpeed*, *RouteCalculatorForBatteryLevel, speedRouteCalculator* (which is designed to quickly find a route), *and LongRouteCalculator* (which takes longer to find the optimal route). A disjoint *GeneralizationSet* concept model indicates that specific classes have no common instances. For example, the incomplete constraint indicates that there are *RouteCalculator* instances that are not instances of either *RouteCalculatorForSpeed* or *RouteCalculatorForBattery*. In constrast, every instance of *RouteCalculator* must be either a *SpeedRouteCalculator* or a *LongRouteCalculator*.

Figure 8. Use of GeneralizationSet to model variability

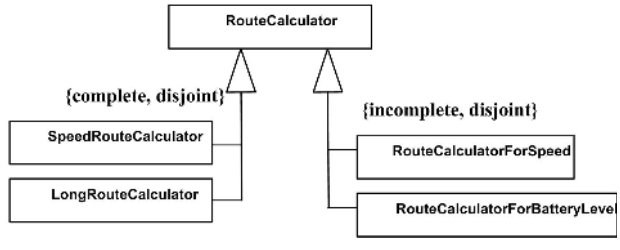


Figure 9. Iterator pattern for position management in the driving system

Pattern

According to Gamma, Helm, Johnson, and Vlissides (1994), "design patterns model the description of communicating objects and classes that are customized to solve a general problem in a particular context." Patterns represent, in a framework, common mechanisms used in the whole of an application domain. Their variability is expressed by connecting the specific classes of the family members to patterns.

In UML 2.0, this mechanism is called *Collaboration*. The pattern is represented as a kind of classifier. The collaboration entity is associated with a set of cooperating entities that play a role in the pattern. Collaboration is a specific view on relations between classifiers. UML patterns are more restrictive than generic patterns because they are focused on structural description, whereas general design patterns may also depict behavioral aspects.

The robot moves by displacement between two positions. To enable this, our framework has to provide a mechanism to manipulate a list of positions. In this case, a position class is not imposed, but instead is considered as a variation point. Thanks to the *Iterator* pattern, the framework proposes a common mechanism for manipulating a position list. To obtain a member of the product family, we can specialize classes and associate a role (see Figure 9).

Standard UML provides many elements (Template, GeneralizationSet, etc.) to describe framework variations and commonalities. Structural entities do not give enough information to describe a system. Designers, therefore, need to describe behavior. The next section explains what UML elements are used to do so.

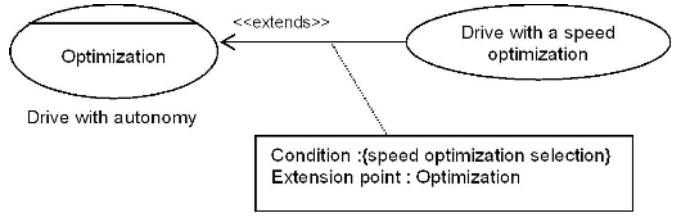
Behavioral Model

This model aims at describing the behavior of applications, either at the system level, for example, through use case diagrams (UCDs), or locally, by means of state machine diagrams (SMDs).

The "Extend" Relationship Between Use Cases

UML proposes to describe system requirements through UCDs. These consist mainly of use case and actor elements. Whereas the former specify functionalities of the system, the latter model elements of the system environment that interact with it. Use cases are associated with actors participating in the functionality, but they may also have the

Figure 10. Extend relationship for autonomous driving



stereotyped interrelationships *«include»* or *«extend»*. The include dependency is used to focus on specific parts of use cases specifying that a use case includes some other use cases.

The extend dependency, which indicates that a use case *may* include another use case, is a better solution from the standpoint of variability modeling. The extend dependency contains a condition. This condition restricts the existence of the extended use case. For example, if the condition is evaluated as false in the specific system design, the extended use case does not appear in the model of this system.

In our case study, the driving system UCD has the following use case: *driveAutomatically*. This use case may present several variations that are modeled using the «extend» dependency relationship. Two variations are described in Figure 10: an automatic driving mode which optimizes speed; and the same mode minimizing energy consumption. These use cases represent variability. In this example, the «extend» relationship has an extension point with a condition. An extension point identifies a point in the behavior of a use case where that behavior can be extended. The extended use case "Drive with speed optimization" appears in the specific system if the condition: "select speed optimization" is true.

There is a lack of precision here for defining the extension point. For example, the reader of this model does not know whether the extension point takes place at the beginning or end of the action sequence contained in the "Drive with autonomy" use case.

Standard UML provides few elements for describing variability. Variations in a structural model have impact on the behavior model. Framework designers thus need to devise state machine and activity diagrams that incorporate variations.

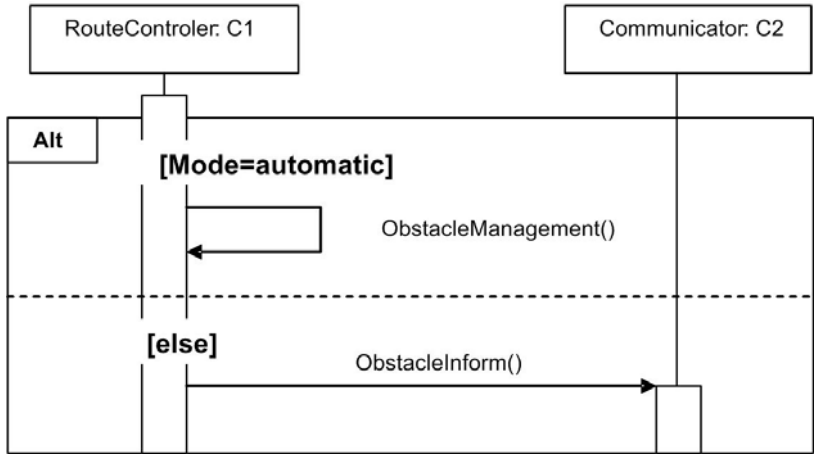
The next section describes the UML elements that indicate interactions between entities based on an analysis of variability expressions.

Interaction Model

Interaction Diagram

In this diagram, an interaction sequence can be grouped into an entity, called a *"CombinedFragment."* UML enables association of constraints with this entity. More specifically, this set of interactions can be specified as alternative or optional. When sets

Figure 11. A CombinedFragment expressing behavior variability in the case of obstacle detection



of interactions are alternative, an associated guard informs the user that only one set of interactions will be executed. When a set of interactions is optional, the associated guard indicates that the set of interactions may or may not be executed. In this diagram, common interactions can be represented without constraints and alternative or optional interactions define variability in the systems family.

These elements are used to specify our drive system. Figure 11 represents an interaction between two instances. The *RouteController* object manages execution; the Communicator object is in charge of sending information to a remote user interface. When the automatic system detects an obstacle, it manages this constraint in an autonomous way. If the system is manual or assisted, it calls for human commands. To indicate these varying behaviors, the *combinedFragment* "ALT" operator (for "alternative") indicates that it is composed of two disjunctions. The first disjunction has a guard "[mode==automatic]" and the second has the operand "else." Using the returned value of the guard, the designer is able to identify the appropriate variability.

Conclusion on the Usability of UML for Variability Modeling

UML provides many elements for describing variability in a framework (templates, generalization). These elements focus essentially on architectural issues. As entities can be variants in a family system, they affect the behavior of that system. UML does not allow variability to be expressed in state-based models. This has led to development of the so-called UML-based methodologies.

UML-Based Methodologies for Product Family Modeling

Building a product line is different from building an application. To design a family of systems, engineers devise a generic model (or framework), then specialize it to obtain a specific system. In such models, it is necessary to describe communalities and variations in the system family. To do so, engineers need the help of a specific methodology. Several approaches now exist for this purpose. They are based on the concept of "profile," which uses standard mechanisms (stereotypes and constraints) to extend/customize the UML language.

This section gives a brief description of UML-based methodologies, which are then analyzed for structural, behavior, and interaction models. Here again, the driving system case study serves to illustrate the explanations.

Outline of UML-Based Methodologies for Variability Modeling

This section depicts the principles and organization of existing UML-based methods for variability modeling.

Catalysis Approach (D'Souza & Will, 1998)

Most of the principles inherent in this approach are applied and used in UML 2.0.

Its purpose is to design a 'generic architecture. The framework is viewed as a basis on which to build members of system families. Emphasis is placed on use of templates, patterns, and components. While the framework does not explicitly describe variation points for the family, variations are obtained by using component properties. This entails combining a basic set of components into different configurations. By specifying these components, Catalysis relies on Component-Based Software Development (CBSD).

This approach is iterative and parallel. It consists of building five layers of abstraction:

1. **Business Model (domain model)** – This model is used to capture term domain rules and business tasks of the target software. The business model step appears in several approaches.
2. **Requirement Specification** – This layer specifies system requirements. Such specification is mandatory for building the complete system. The requirement specification step is used to refine specifications and to give an idea of the system components.
3. **Component Design** – This layer provides a high-level specification of major system components along with requirements for each.

4. **Object Design** – For each component, this layer describes the set of classes and how they work together.
5. **Component Kit Architecture** – This layer describes common elements of the component collection. It is used to group components into product families.

UML-F (Fontoura, Pree, & Rumpe, 1999; Fontoura, Braga, Moura, & Lucena, 2000; Fontoura & Lucena, 2001; Fontura, Pree, & Rumpe, 1999)

In 2001, the members of this research team noticed that the UML standard did not provide appropriate constructs to model frameworks. There were no indications in UML design diagrams of variation points and instantiation constraints. The researchers then applied a UML mechanism to add extensions in the standard specification. This work led to the UML-F profile.

According to the authors of UML-F, framework building also requires new methodological tools and concepts. Their methodology is intended to permit design of an application from the start as a family. It is based on the cluster cycle process specified by Meyer (Bertrand, 1990) and includes the following steps for systematic and efficient development:

1. **Identification of key domain abstractions** – This step calls for use of the lass-Responsibility-Collaboration (CRC) methodology and for determination of abstract system entities – classes and interfaces. The authors then propose to design class families (class sets that implement an interface), class teams (class sets that collaborate), and subsystems (entities that encapsulate class teams).
2. **Definition of flexibility requirements (variation points)** – Once the main framework entities are identified, the next step consists of determining variation points. This is done with the help of a domain application expert.
3. **Design of refinement and transformation into an architecture** – In this step, engineers again analyze system architecture with the help of an expert. They examine class distribution and identify patterns in order to improve flexibility. This is the point at which the UML-F profile is used. To deal with system complexity, authors have added the notion of framelet. Framelets are groupings of patterns and variation points intended to help users design frameworks. The authors consider a framelet as an abstract view of the framework.
4. **Adaptation of the framework** – To enable use of the product line approach, designers must specialize the framework (i.e., add and modify specific entities in the model). To guide them in performing this task, the authors propose the concept of implementation case. Implementation cases describe an aspect of the application instantiation process by specifying how a component, an architectural feature or a functionality for an application in the framework domain can be implemented using the constructs provided by the framework.

Kobra Approach (C. Atkinson et al., 2002)

Kobra is a product line approach relying on Component-Based Software Engineering. It represents an object-oriented customization of the PuLSE method (Bayer et al., 1999). The infrastructure construction phase of PuLSE corresponds to Kobra's framework engineering activity.

Like UML-F, this approach calls for designing a framework with variations at system level as early on as possible in development. It assists the user in choosing suitable variation points from among those available in a framework — a feature afforded by designing a decision model.

To permit design of a component-based framework, the authors have defined their own component concept known as a Komponent. This entity merges the UML concepts package and the component. In the first stages of the approach, Komponents are viewed as packages, then, after several refinements, they become components in the UML1.x sense (implementation parts such as binary files[2]).

The Kobra approach then requires the following to obtain a product line:

1. **Variability identification** – Komponent creation and variability identification take place at the same time. The set of concepts used to express variability in a model is small. Variance is modeled everywhere using the «variant» stereotype. Variability in the Kobra process can be conveyed at various granularities: the Komponent entity or method can be variable. The repercussions of these variances are found in the structural, behavior, and interaction models.
2. **Decision modeling** – Knowledge of the application domain is an important source of dependencies and constraints for the decision model. This construction is complex and requires the assistance of an expert. All decisions are postponed to cope with existing variations, which are identified during system analysis. For each variable element, a simple decision is created as a question associated with the modification to be made to the system. The model obtained the results in a checklist in which each question can be interdependent.
3. **Component packaging** – This step consists of specializing the framework and thus requires the user to scan the decision model. Each answer to a question implies modifications. The application model is complete once the decision model has been fully scanned. It then remains for the user to implement and deploy the components obtained on the relevant hardware. Nothing specific is proposed by the authors to perform these final tasks.

CAFÉ

CAFÉ is an ITEA[3] project that was itself an extension of the ESAPS[4] project. The ESAPS project allows definition of family system concepts. It defined the concept of variability

and communalities to describe a systems family. It proposes a sequence of steps for building systems: domain analysis (identification of requirements), domain design (design of the system), domain implementation, and component paradigms, which are used throughout the process.

The CAFÉ project mainly aims to bring ESAPS-defined concepts to maturity so that they can be applied to tangible projects, by using them to develop methods and procedures. The results of the project, which will encompass knowledge of methods and procedures, will be used for designing tools and concrete applications.

All product line approaches have the same principle. They determine the entities and system variations needed to build a system, then specialize the system to obtain an application. In these approaches, several profiles are adopted to express model variations.

Work performed as part of the CAFÉ project has been geared to integrating the variation concept of FODA into the UML formalism. FODA is a product line approach. To represent a product family, it uses the concept of features (Kang et al., 1990):

"Features are the attributes of a system that directly affect end-users. The end-users have to make decisions regarding the availability of features in the system, and they have to understand the meaning of the features in order to use the system."

There are several relationships available to express variability between these features.In the CAFÉ project, UML elements may be mandatory, optional, or alternative. This CAFÉ project explicitly describes the relationships between variations using OCL rules.

All of the discussed approaches also use UML 2.0. The following section assesses the structural elements used by each to describe variation in a product family.

Variability In structural Models

Catalysis

The Catalysis approach is mainly architecture-oriented. It made use of parameterized packages even before this notion was standardized by the OMG. Catalysis also introduces the concept of package inheritance. The original package can be contained in the framework and represents the commonality of the application domain. For each application, said package is specialized to meet specific requirements. Catalysis also introduces many other concepts to describe reusable mechanisms such as templates or patterns. The ultimate goal is to obtain components, which, as already seen, are an ideal means for expressing variability.

UML-F

The UML-F profile adds a lot of information to standard UML. In the structural model, its purpose is to help the user distinguish common elements. It therefore facilitates variability assessment. This entails using a set of tags for classes or methods to identify those that are fully defined. Other tags are used to visualize the elements added by an inheritance relationship (see Figure 12).

Several tags are used to define properties of design elements. They indicate whether or not an element belongs to the specific application or framework. They also may indicate whether that element can be modified for framework specialization purposes.

In Figure 12, the © sign indicates that the specification is complete and the sign identifies any new properties that are added in a specialized class. The gray rectangle indicates that the properties are new in the class that features them.

UML-F does not make use of the template concept. The authors of this approach developed it for JAVA language, in which there are no templates. To compensate, the appropriate elements are therefore stereotyped «template» and «hook» (see Figure 13). The hook stereotype is applied for operations that potentially have different implementations. As shown in Figure 13, the *record()* method can be implemented in various ways, depending on the information recorded.

The usefulness of a hook stereotype is questionable in cases where UML template entities are available. However, this approach provides a means for expressing template mechanisms in implementation languages that do not support templates.

Figure 12. Class diagram for route calculus

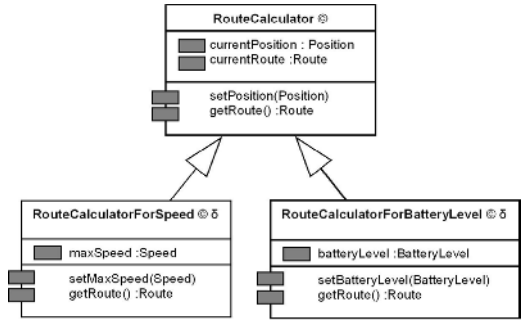


Figure 13. Expression of variability using a template and hook stereotype in the InformationRecorder class

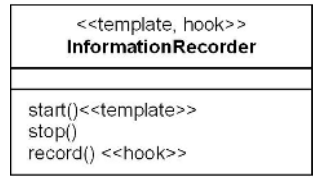
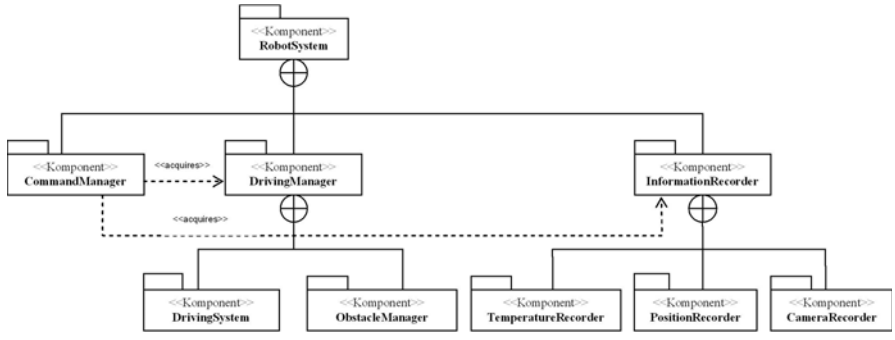


Figure 14. Kobra komponent tree



To express the pattern reuse mechanism, the profile provides a set of stereotypes that describe the nature of patterns (factory, singleton, etc.).

Kobra

Kobra is a component-based approach whose purpose is to describe variant elements in its architecture.

To enable use of a component approach, Kobra organizes its architecture as a komponent tree (see Figure 14). In this view, variations do not appear, but information is provided about relations between Kobra components. At the root of the tree, there is a komponent that represents the system. Subkomponents then represent system constitutive entities. Components also have relationships with type indications about their links: creation, acquisition. A creation link indicates that a component A obtains an instance of a component B by creating it.

In this methodology, variations are clearly expressed. The Kobra components themselves, as well as any of their methods, can be variants. Such variants can be seen in a structural view centered on each component (see Figure 15). The targeted komponent of the view features the «subject» stereotype. In the resulting diagram, relationships between classes and komponents are visible, and variability can be quickly visualized. The variable methods are tagged «variant» and the classes or komponents are stereotyped by the «variant» keyword. In a Kobra diagram, variant elements do not have

Figure 15. variability in a Kobra class diagram

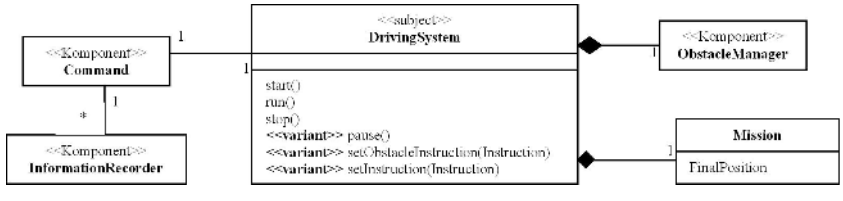
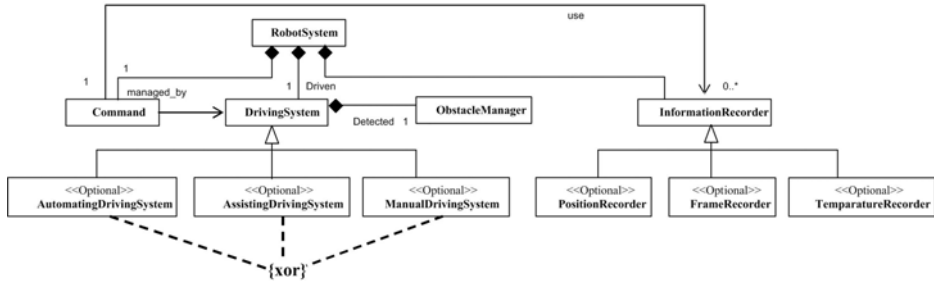


Figure 16. Variability in a class diagram (Ziadi et al., 2003)



constraints. Relations between variant elements are defined in textual form in the decision model.

CAFÉ

The structural model places emphasis on variations between architectural elements.

This model is based on the FODA principle specially developed to facilitate analysis and produce a graphical representation of domain application features.

As a means for expressing these qualities (mandatory, alternative, optional) in UML formalisms, the CAFÉ project (Speck, Clauss, & Franczyk, 2001; Ziadi, Hélouët, & Jézéquel, 2003) suggests the addition of stereotypes and OCL constraints.

This CAFÉ project uses the UML structural diagram to represent features. Each object is a feature of the system that can have a mandatory, optional or alternative stereotype. Objects are connected by aggregation links with OCL constraints to express certain dependencies between features. The advantage of this approach is that OCL rules provide information about dependencies between variations.

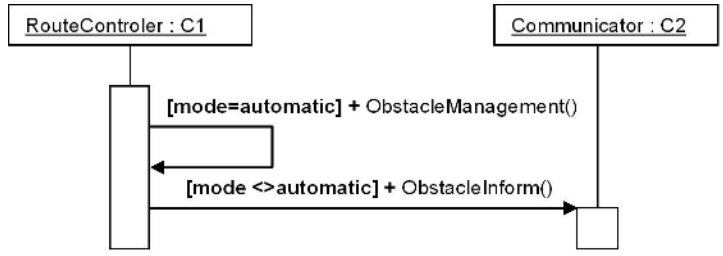
In our example (see Figure 16), based on the work of (Ziadi et al., 2003), an "xor" OCL constraint is used to express an alternative between two features.

To conclude this section on variability expression in the structural domain, it is noteworthy that all the approaches mentioned provide mechanisms for describing variations. They enable description of a model with fine-grain variants.

For example, in the Kobra approach, operations can be variants. It is difficult, however, to describe relationships between such precise variants. This approach therefore provides OCL rules to explain such relationships.

The next section examines variation elements available for interaction models under the different approaches described and indicates whether such elements can make up for those lacking in UML.

Figure 17. Variability expression with UML-F for a route control sequence diagram



Variability in Interaction Models

Catalysis

This approach has no specific mechanisms for expressing variability in the interaction model.

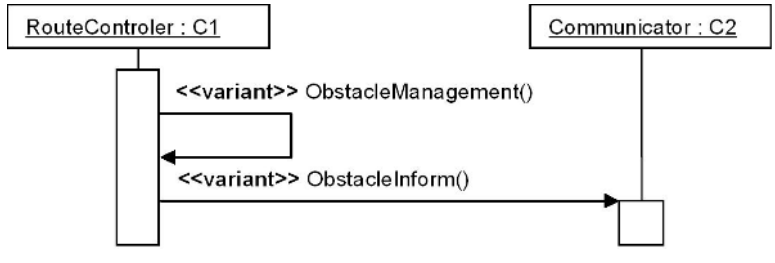
UML-F

UML-F provides means for describing variation in a sequence diagram. Use of conditions at message output in combination with specific tags ensures definition of optional messages and specifies alternatives. It also is possible to define a sequence of messages produced when a specific message is sent. Conditions then clearly indicate the effects of choosing a variation.

In Figure 17, the tag '+' means that the message can be optional. In our example, it can be used with or without a condition.

The main drawback of this approach is that there are no clear and explicit links between conditions defined in the sequence diagram. Defining/changing conditions can thus be very difficult since it is then necessary to check all other conditions defined in the sequence diagram to ensure consistency.

Figure 18. Variability expression in the Kobra route control sequence diagram



Kobra

In Kobra, a method can be a variant, which means that behavior is likewise subject to variants. The resulting impact can be visualized in the diagram of interaction. Operation sequences in the interaction diagram also can be stereotyped by the word "variant" (see Figure 18). This stereotype allows the user to quickly visualize variations, but the same problem occurs as for structural views, that is,. information about relationships is not indicated. For this reason, it is difficult to determine the specific interaction diagram of a particular system.

CAFE

Combined fragments defined in UML2 sequence diagrams are used to specify variability points within interaction models. In this case, conditions define whether a combined fragment can be executed or not.

The main drawback here is similar to the one encountered in the UML-F approach. If a variation also is changed or added, the designer has to review all other conditions defined in the sequence diagrams for the sake of consistency.

These approaches all provide mechanisms for describing interaction variability; but the diagrams are difficult to construct. Where conditions of variation are indicated, it is difficult to ensure consistency between diagrams. Designers must check all the conditions for each diagram.

The next section examines the means available for expressing behavioral variability.

Variability in Behavior Models

Catalysis approach

To express behavioral variability, Catalysis proposes a diagram based on the actors, use cases and objects of a system. This diagram allows the association of objects with their use cases and actors.

Figure 19. Diagram of action and object constituents

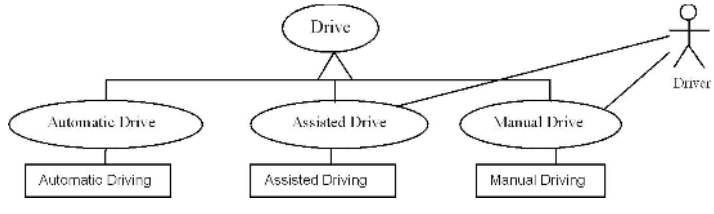
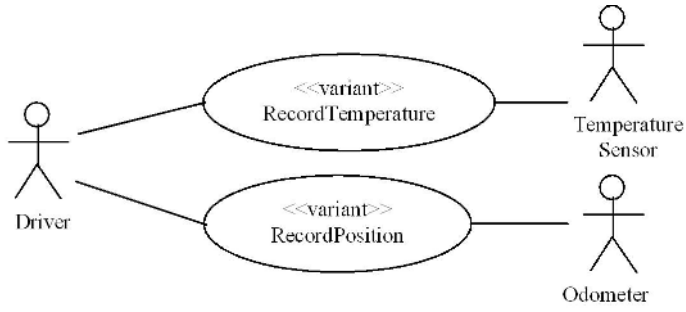


Figure 20. Use case diagram for an information recorder: two possible variations



To express the features of a product family, Catalysis adds variant actors and variant use cases by means of inheritance links. At this level, the object does not have the property of being inherited. An entity becomes a template package and finally a component, by refinement. It gives an idea of the future system component. Using the Catalysis diagram, designers can associate variable entities with variable functionalities. For this reason, the interaction diagram itself does not inform the user about variance. Instead variants are associated with each future component.

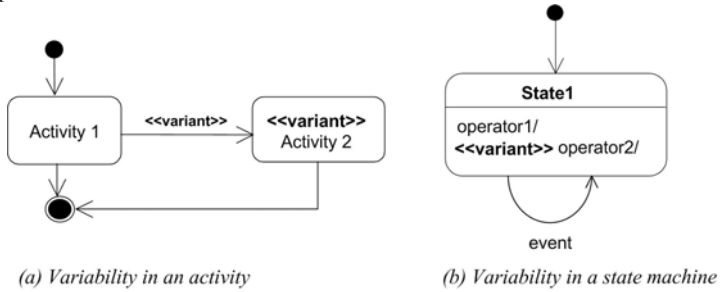
In Figure 19, there are three optional variations: *automaticDriving*, *assisted driving*, and *manual driving*.

Kobra Approach

In this approach, variability is expressed in the behavior model.

Kobra expresses the variant functionality from the very first stages in development. It therefore appears in the use case diagram. The relevant use cases are stereotyped "variant" (see Figure 20). Since Kobra system design is based on use cases, it is easier for the designer to describe variant elements in the future framework architecture.

Figure 21. Variability expression in the activity and state-machine diagrams under the Kobra approach



(a) Variability in an activity diagram

(b) Variability in a state machine diagram

The Kobra approach provides means to express variability of object operations. This fine-grain variability has impact on activity diagrams and state machine diagrams. In the activity diagrams (see Figure 21), activities and subactivities can have a *«variant»* stereotype. The branches that reach these elements also can be stereotyped *«variant»*. In the state machine diagrams (see Figure 21), the states are not variant but the events issuing from class operations can be stereotyped *«variant»*. The state machine therefore always has the same shape. Variant elements do not influence the global behavior of a class.

This approach allows description of variations and their consequences for system behavior. For example, a variation can add an activity to an activity diagram. On the other hand, there are no indications about relationships between variations in the behavior model. Therefore, it is difficult to get a precise idea of the behavior of a specific system. Alternative variations, for example, are not clearly expressed.

CAFÉ

In this approach, both *«variant»* or *«optional»* stereotypes can be applied to use cases and actors of the use case diagram. Relationships (e.g., alternatives) between variations are not depicted. The stereotyped *«extend»* relationship between use cases also may be applied. Thanks to these stereotypes, variations in a family product can be preanalyzed in use case diagrams. Designers can then quickly visualize the different variations of functionalities provided by a system family, in order to make choices for a specific application.

The approaches described all have few elements available to describe variability in the behavior model. Variations in statemachines, for example, only are indicated by Kobra. Note that the management of variations in a behavior model is difficult. Behavior variations in fact depend on structural and interaction elements. This is further complicated by the fact that variations are of different types—alternative, optional, and so forth. Therefore, designers require suitable tools for maintaining these variations and ensuring the consistency of diagrams.

State-of-the-Art Variability Modeling

Table 2 provides an overview of the various tools offered by UML-based methodologies designed to support a product line approach. More specifically, it shows how these methodologies manage variability description in UML models.

All of these approaches provide means for expressing variability. Catalysis and UML-F use mechanisms provided by UML. Kobra and CAFÉ use a profile to directly express variations. The expression of variability in the behavioral and interaction models is not well developed. Only Kobra and CAFÉ have the necessary mechanisms. CAFÉ uses UML 2.0 sequence diagrams, and Kobra uses the "variant" stereotype.

Only Kobra enables separation of concerns between variability expression in models and choices to select variations. It can express variability in a system and provides a decision

Table 2. An overview of the various tools offered by UML-based methodologies

	Variability in a structural model	Variability in an interaction model	Variability in behavioral model	Decision Model	Support Tool
Catalysis	Pattern template + Component. These mechanisms are present in UML2.0	Nothing specific. Uses UML1.1.	Extension of UML use case diagrams. Use of inheritance links between use cases.	Extension of UML use case diagrams. This diagram enables the user to choose components according to use case.	SmartDraw seems to support this approach. This tool is used to draw Catalysis diagrams.
CAFÉ	Apply <<optional>> stereotype to entity with OCL constraints in the class diagram Reuses features of FODA approach.	UML2.0 combined fragment of sequence diagrams. Expression of alternative, optional variations.	Apply <<variant>> and <<optional>> stereotypes to actor and use case. Its use also extends the relation to express variability.	Nothing specific. In Ziadi's approach, the user can read the class diagram to specialize the framework. OCL constraints help the user to choose variations.	No
UML-F	Pattern template This approach does not provide new mechanisms for expressing variability	Stereotyped action in sequence diagrams to model repetition or optional features of message	Nothing specific. Only UML1.4 is used.	Instantiation case depicted via textual scenarios	No
Kobra	Stereotype <<variant>> on components and methods. These elements may or may not be present in a specific application	Apply <<variant>> stereotype to message of sequence and collaboration diagrams. There is consistency with other diagrams.	Apply <<variant>> stereotype to use case transitions of activity diagrams and state/transitions of state-machine diagrams	The model decision I made with an expert. The decision is check-list	A prototype exists It runs with rational. Users can use stereotype Kobra, but it does not allow an automatic specialization of the framework.

model to indicate relationships between variant entities. Other approaches combine descriptions of variations with their relationships. With Kobra, the designer knows all the relationships between variant entities and can easily specialize the framework. The Kobra decision model is built from all specifications of family systems. In the future, it may be of interest to generate a decision model from the framework.

Few tools are proposed by these approaches. Some tools, however, are available to manipulate a framework. In general, designers need to adapt an approach to a CASE tool. It is probable that other tools will be developed in the future to help designers specialize a framework. Such tools will be able to specialize variations according to designer needs.

Approaches Under Development

CEA-List also is working on software product line concepts. To this effect, it is participating in the ITEA project FAMILIES (http://www.esi.es/en/Projects/Families/famMain.html).

Outline of the FAMILIES Project

Members of the project consortium have now been investigating and developing system family technologies for seven years, and the experience gained so far has been quite

significant. In particular, they have been involved in ITEA projects ESAPS and CAFÉ, which focus on a similar subject. This has led to a recognized community of European experts on System Family Engineering. The FAMILIES project aims at expanding the community and consolidating results into a fact-based management for the practices of FAMILIES and earlier projects. It also plans to explore fields that were not covered by previous projects in order to complete the proposed framework for product family support.

FAMILIES is the finalizing stage for both ESAPS[5] and CAFÉ[6] European projects. It focuses on maturity, institutionalization, business relevance, standardization, and dissemination of product families.

Conclusion

Distributed information systems feature an increasing number of functionalities and require longer and longer development times. At the same time, heightened competition is pushing industrialists to always get their systems out first. The resulting products have long service lives and must undergo evolutions. One way of keeping them in service, while enabling gradual addition of new functionalities, is to create a "system family" framework (McIlroy, 1968). Each subsequent version of the system is then viewed as a member of the same family. In this chapter, we described the different approaches available for doing so.

There are many formalisms for describing the framework, or generic architecture, of a system family. Since UML is now the de facto standard for modeling, many of the approaches discussed here employ UML to express product line requirements.

One of the tenets of a product line approach is that it be able to express differences between family members in a model. These differences can be expressed by variations or "hot spot" elements in the framework.

UML now provides more and more mechanisms for expressing such variability in design models. To quote one example, template packages and variability in interaction models appeared recently in UML 2.0.

Several approaches now enable extending UML, to account for greater variability. As seen in this chapter, variability may exist at several granularities: package, class, and method. It also may be assigned a type, for example, optional, mandatory, or alternative.

Variability introduction makes models more complex, and it then becomes difficult to assess the impact of variations on the framework. To deal with such problems, system family approaches like Kobra (Atkinson, Bayer, & Muthig, 2000) use the concept of decision model. This is an artifact that guides user choice of variable entities during the framework specialization phase. Most of these approaches rely on the user to construct the decision model.

The purpose of our particular approach is to help engineers construct software in the real-time domain by automating as many analysis steps as possible. This entails adapting product line approaches to expressing differences in distributed real-time systems. In the

resulting methodology, emphasis is placed on the construction and execution of decision models.

Our future research will concentrate on extending UML to expressing variability in real-time systems. During the refinement of our model, several OCL rules will be written to indicate relationships between variations. This will give rise to a tool for both constructing the decision model and helping designers specialize the framework to obtain a specific system.

References

Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., et al. (2002). *Component-based product line Engineering with UML.* Addison Wesley.

Atkinson, C., Bayer, J., & Muthig, D. (2000). *Component-based product line development: The KobrA approach.* Paper presented at the the First Software Product Line Conference.

Batory, D., Chen, G., Robertson, E., & Wang, T. (2000). Design wizard and visual programming environment for GenVoca generators. *IEEE Transaction on Software Engineering,* 26.

Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., & Widen, K.S T. (1999). *PuLSE: a methodology to develop software productline.* Paper presented at the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99), Los Angeles.

Becker, M., Geyer, L., Gilbert, A., & Becker, K. (2001). *Comprehensive variability modelling to facilitate efficient variability treatment.* Paper presented at the Software Product Family Engineering, Bilbao, Spain.

Bertrand, M. (1990). Lessons from the design of the Eiffel librairies. *ACM, 33*(9).

Bézivin, J., Dupé, G., Jouault, F., Pitette, G., & E.Rougui, J. (2003). *First experiments with the ATL model transformation language: transforming XSLT into XQuery*. Paper presented at the OOPSLA 2003 Workshop, Anaheim, California.

Bosch, J. (2001). *Software product lines: Organizational alternatives.* Paper presented at the 23rd International Conference on Software Engineering.

Christian Bunse, C.A. (2001). *Implementation of component based systems by semantic refinement and translation steps.* Paper presented at the WTUML: Workshop on Transformations in UML, Genova, Italy.

Clauss, M. (2001). *Generic modeling using UML extensions for variability*. Paper presented at the An OOPSLA Workshop 2001, Tampa Bay, Florida, USA.

Clements, P., & Northrop, L.M. (2001). *Software product lines: practices and patterns.* Addison Wesley.

D'Souza, D., & Will, A. (1998). *Catalysis: Objects, framework and components in UML.* Addison Wesley.

Fontoura, M., Pree, W., & Rumpe, B. (1999). *UML-F: A Modeling Language for Object-Oriented Frameworks (No. Technical Report TR-613-99).* Computer Science, Princeton University.

Fontoura, M. F., Braga, C., Moura, L., & Lucena, C. J. (2000). *Using domain specific languages to instantiate object-oriented frameworks.* Paper presented at the IEE Proceedings - Software.

Fontoura, M. F., & Lucena, C. J. (2001). Extending UML to improve the representation of design patterns. *Journal of Object-Oriented Programming (JOOP), 13*(11), 12-19.

Fontura, M., Pree, W., & Rumpe, B. (1999). *UML-F: A modeling language for object-oriented frameworks. (No. Technical Report TR-613-99).* Computer Science, Princeton University.

Franch, X., & Ribo, J. M. (1999). *Using UML for software process modelling.* Paper presented at the UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, For t Collins, CO.

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

Gérard, S., Terrier, F., & Tanguy, Y. (2002). *Using the model paradigm for real-time systems development: ACCORD/UML.* Paper presented at the OOIS'02-MDSD.

Harsu, M. (2002). *FAST product-line process (No. 29).* Institute of Software Systems,Tampere University of Technology.

Kang, K. C., Cohen, S.G., Hess, J.A., Novak, W.E., & Peterson, A.S. (1990). *Feature-oriented domain analysis (FODA).* No. CMU/SEI-90-TR-21 ESD-90-TR-222. Carnegie Mellon University.

McIlroy, M.D. (1968). *Mass-produced software component.* Paper presented at the NATO SCIENCE COMMITTEE, Garmisch, Germany.

OMG. (2002). UML 1.4 with Action Semantics.

OMG. (2003). UML 2.0 OCL 2nd revised submission (No. ad/03-01-07).

Parnas, D. L. (1979). Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering, SE-5,* 128-137.

Pollet, D., Vojtisek, D., & Jézéquel, J.-M. (2002). *OCL as a core UML transformation language.* Paper presented at the WITUML, Malaga, Spain.

Speck, A., Clauss, M., & Franczyk, B. (2001). *Concerns of variability in "bottom-up" product-lines.* Paper presented at the German GI Workshop Aspect-Oriented Programming, Universität Paderborn.

Svahnberg, M., & Bosch, J. (2000). *Issues concerning variability in software product lines.* Paper presented at the the Third International Workshop on Software Architectures for Product Families, Berlin, Germany.

Tessier, P., Gérard, S., Mraidha, C., Terrier, F., & Geib, J.-M. (2003). *A component-based methodology for embedded system prototyping.* Paper presented at the 14th IEEE International Workshop on Rapid System Prototyping, San Diego, CA.

Ziadi, T., Hélouët, L., & Jézéquel, J.-M. (2003). *Modélisation de ligne de produits en UML*. Paper presented at the Langages et Modèles à objets (LMO'03).

Ziadi, T., Jézéquel, J.-M., & Fondement, F. (2003). *Product line derivation with uml*. Paper presented at the Software Variability Management Workshop, University of Groningen Departement of Mathematics and Computing Science.

Endnotes

1 OCL is language defined by OMG to add constraints in UML models: http://www.omg.org/

2 In the UML2 context, these are equivalent to component artifacts. Komponents of the Kobra approach are then fully in keeping with the Component concept redefined in UML2.

3 ITEA (Information Technology for European Advancement) is an eight-year strategic pan-European program for advanced precompetitive research and development in embedded and distributed software.

4 ESAPS = Engineering Software Architectures, Processes and Platforms for System-Families : http://www.esi.es/en/Projects/esaps/esaps.html

5 http://www.esi.es/en/Projects/esaps/esaps.html

6 http://www.esi.es/en/Projects/Cafe/overview.html

About the Authors

Hongji Yang is a professor and head of the Software Engineering Division in the School of Computing at De Montfort University, UK. He received a BSc and an MPhil in computer science from Jilin University, China, and a PhD in computer science from Durham University, UK. His research interests include software engineering and distributed computing. He served as program co-chair at the IEEE International Conference on Software Maintenance (ICSM'99), program co-chair at the IEEE International Workshop on Future Trends in Distributed Computing Systems (FTDCS'01), and the program chair at the IEEE Computer Software and Application Conference (COMPSAC'02).

* * *

Marcus Alanen received his MSc in computer engineering at the Åbo Akademi University in Turku, Finland (2002). Currently, he works as a PhD student at the Department of Computer Science at Åbo Akademi University.

Gabriel Baum (gbaum@info.unlp.edu.ar) is full professor at Universidad Nacional de La Plata (UNLP), Argentina. He teaches courses in functional programming and formal languages. He has published a large number of papers of his work on formal methods including specification and derivation of programs. He is leader of several national and international research projects. Since 2002, Professor Baum is the president of SADIO, the Argentinean Society of Computer Science. For more information, visit http://portal-lifia.info.unlp.edu.ar/~gbaum.

Cornelia Boldyreff has more than 25 years experience in software engineering and has been a professor of software engineering at the University of Lincoln, UK, since January 2004 where she leads the Distributed Software Engineering group. Her research spans both distributed systems and software engineering. Currently, she is focused on collaborative software development within the EU Framework 5 GEneralised eNvironment for procEsS management in cooperative Software Engineering (GENESIS) project and the UK EPSRC Collaborative Determination, Elaboration, and Evolution of Design Spaces (CoDEEDS), collaborative learning via the Grid within the LeGE-WG project, and system evolution within the Web Site Evolution project. The group is currently developing the Open Source Component Artefact Repository within GENESIS and developing support for collaborative design teams within CoDEEDS.

Rodrigo E. Caballero is a staff research engineer with the United Technologies Research Center in East Hartford, Connecticut (USA). His specialization at UTRC is in the area of Controls and Embedded Systems. Mr. Caballero is a PhD student in computer science at the University of Connecticut. The focus of his PhD research is in software reusability and refactoring for component-based systems.

Ana Isabel Cardoso is an invited professor in the Engineering and Mathematical Department at Madeira University, Portugal, and a PhD student in information technology and computer engineering at the Technical University of Lisbon, Portugal. She has been a project manager in Portugal Telecom for the last 20 years. Her research interests include complex systems, software process, and metrics.

Chih-Hung Chang received his MS and PhD degrees in information engineering and computer science from Feng Chia University, Taichung, Taiwan (1997 and 2004, respectively). He is currently an assistant professor in the Department of Information Management of Hsiuping Institute of Technology, Taiwan. His areas of research interest include software maintenance, software reengineering, design pattern, component-based software engineering, reuse, and software process document formalization.

William C. Chu is a professor and the chairman of the Department of Computer Science and Information Engineering at Tunghai University, Taiwan. From 1994 to 1998, he was an associate professor at the Department of Information Engineering and Computer Science at Feng Chia University. He was a research scientist at Software Technology Center of the Lockheed Missiles and Space Company, Inc., where he received special contribution awards in both 1992 and 1993 and the PIP award in 1993. In 1992, he also was a visiting scholar in the Department of Engineering Economic Systems at Stanford University, where he was involved in projects related to intelligent knowledge-based expert systems. His current research interests include software engineering, embedded systems, and E-learning. Dr. Chu received his MS and PhD degrees from Northwestern University in Evanston, Illinois, in 1987 and 1989, respectively, both in computer science. He has edited several books and published over 100 refereed papers and book chapters, as well as participated in many international activities, including organizing many

international conferences. Dr. Chu has received many research awards and funded research grants. His e-mail address is chu@csie.thu.edu.tw.

Paul Crowther completed his PhD at the University of Tasmania, Australia (1999) where he also was a lecturer. He is currently at Sheffield Hallam University, UK. His research interests include requirement elicitation and intelligent agent-based applications. He has been working on the European MOBIlearn project for the last two years. Sheffield Hallam is one of more than 20 European partners working on this collaborative project.

Steven Demurjian is a full professor and associate department head of computer science & engineering at The University of Connecticut, USA, with research interests of secure software design using UML and aspect-oriented programming, security requirements specification and assurance, RBAC/MAC models and security solutions for UML, distributed environments and XML documents, and reusability and refactoring for component-based systems (UML and Java). Dr. Demurjian has more than 100 publications, in the following categories: 1 book, 1 edited book, 8 journal articles, 24 book chapters, and 72 refereed articles.

Damiano Distante is a post-doc at the Research Centre on Software Technologies (RCOST) in the Department of Engineering of the University of Sannio in Italy, and an instructor of database systems in the Faculty of Engineering of the University of Lecce. He has a PhD from the University of Lecce and spent part of his doctoral studies at the Florida Institute of Technology. His research interests include conceptual modeling of Web applications, Web transaction design and reengineering, Web engineering, and recently, service-oriented system engineering. He is the program chair for WSE 2005.

Felix Eickhoff graduated from The University of Connecticut with a MS in computer science & engineering in August 2002 with a research focus on UML and software reusability. In May 2004, he graduated from the University of Stuttgart (Germany) with a German Diploma in software engineering and a research focus on distributed systems, especially mobile ad hoc networks and sensor networks.

Rainer Frömming works as a senior consultant at the 4Soft GmbH in Munich, Germany. In 1999, he graduated in computer science at the Technical University of Munich. Since then, his working experience involves mainly object-oriented software development, software development methodology, project management, and resource management. Mr. Frömming was successfully leading several large software development projects in the engineering process management domain. Concurrently, he was taking part in the state-aided Software Quality Rates Maturity (SoQrates) project, aiming at the improvement of development processes by performing a ISO15504/SPiCE assessment.

Jean-Marc Geib is full professor of computer science at the University of Lille (France). He is currently leading the Software Engineering Group on Objects and Components

(GOAL) of the Computer Science Laboratory of the University. This group tries to contribute to the design and implementation of future platforms for component-oriented distributed computing. One of the current projects, named OpenCCM, is about designing a full software process to product and exploit reliable components following the Corba Component Model (CCM) of the Object Management Group (OMG). Jean-Marc Geib received his PhD from the University of Lille in 1989 for his work on distributed operating systems, and his "Habilitation à Diriger des Recherches" in 1993 for his contribution to object-based distributed computing. Mr. Geib is also the director of the Computer Science Laboratory of the University of Lille and the creator of IRCICA, a French virtual laboratory for mixed basic researches on hardware and software components.

Sébastien Gérard has a doctorate in computer science. He also graduated in 1995 from French Superior School of Mechanics and Aeronautics at Poitiers (ENSMA) as a mechanical and aeronautics engineer. He worked for one year at Laboratory of Scientific and Industrial Computer Science of Poitiers (LISI), where he had interest in defining a methodology for the measurement of the worst case execution time of real-time application. Today, he is researcher at CEA - French Atomic Energy Agency (LIST) in the LSP Group (Software for Process Safety) where he leads the research theme: "Model-Based Software Engineering for RT Systems." He also is strongly involved in UML standardization working on UML within the U2P group and being part of the FTF of the real-time UML profile and supporter of the incoming UML profile for QoS and fault tolerance. Finally, he is the main initiator of this series of workshop dedicated to the development of distributed, real-time, and embedded systems using the model-driven paradigm in the one hand. And he was co-organizer of the international summer school on MDD for DRES located in France in September 2004 (www.ensieta.fr/mda).

Lars Grunske is a research assistant at the Department of Software Engineering and Quality Management of Hasso-Plattner-Institute for Software Systems Engineering. He received a diploma in informatics from the Technical University of Berlin and a diploma in computer engineering from the Berufsakademie, Berlin. His research interest includes software/system architectures, quality characteristics and requirements, architecture transformation, and architecture evaluation. In his recently finished PhD thesis, he explores the problem of how to improve quality characteristics at an architectural level.

Rui Gustavo Crespo is an assistant professor in electrical engineering and computers at the Technical University of Lisbon, Portugal. He received his PhD in information technology and computer engineering from the Technical University of Lisbon. His research interests include protocol specification, feature interaction, and software quality models.

Xudong He received a BS and MS in computer science from Nanjing University, China, in 1982 and 1984, respectively. He received a PhD in computer science from Virginia Polytechnic Institute & State University (Virginia Tech) in 1989. He joined the faculty in the School of Computer Science at Florida International University (FIU) in 2000, and is

an associate professor of the School of Computer Science and the director of the Center for Advanced Distributed System Engineering. Prior to joining FIU, he was an associate professor in the Department of Computer Science at North Dakota State University since 1989. His research interests include formal methods, especially Petri nets, and software testing techniques. He has published over 70 papers in the above areas. Dr. He is a member of the Association for Computing Machinery, and a senior member of the IEEE Computer Society.

Jan Jürjens leads the Competence Center for IT-security at the Software \& Systems Engineering chair at TU Munich (Germany). Holding a Doctor of Philosophy in Computing from the University of Oxford, he is the author of "Secure Systems Development with UML" (Springer-Verlag, 2004) and numerous publications on computer security and safety and software engineering. He is the founding chair of the working group on Formal Methods and Software Engineering for Safety and Security within the German Society for Informatics (GI). He is a member of the executive board of the Division of Safety and Security within the GI, the executive boad of the committee on Modeling of the GI, the advisory board of the Bavarian Competence Center for Safety and Security, the working group on e-Security of the Bavarian regional government, and the IFIP Working Group 1.7 "Theoretical Foundations of Security Analysis and Design".

Peter Kokol is a full professor at the University of Maribor, Slovenia, and obtained a PhD from the University of Maribor. He is the head of Laboratory System Design and head of Centre for Medical Information. Since 1997, he wass the head of Research Institute at the University College of Nursing Studies and from February 2001 also the dean for Research. Since January 2002, he is the director of the Independent Centre for Interdisciplinary and Multidisciplinary Studies and Research. He has written over 300 technical and research papers published in recognized international journals and major conferences and co-authored some textbooks. He was the general and program chair of some major conferences, had numerous invited presentations, and won several best paper awards. His main research interests are intelligent systems, complex systems, system and chaos theory, software quality and metrics, and medical and nursing informatics.

Phyo Kyaw has a BSc in computer science from the University of Sunderland and an MSc in computer science from the University of Durham, UK, where he is currently studying for a PhD. His research interests include distributed software development and collaborative software engineering.

Janet Lavery has a BSc and a research MSc from the University of Durham, Department of Computer Science. She has worked on a range of research projects including Generalised Environment for Process Management in Co-operative Software Engineering (GENESIS), a project focusing on issues surrounding collaborative working environments and the Institutional Secure Integrated Data Environment (INSIDE), an investigation into the issues surrounding the development of managed learning environments for higher education. In addition, she has contributed to the Learning Grid of Excellence —

Working Group (LeGE — WG), whose aim is to facilitate the establishment of a European Learning Grid Infrastructure by supporting the exchange of information and creating opportunities for closer collaboration between the different stakeholders. Currently, she is taking a year off from structured research to teach in order to gain practical experience with student learning issues.

Mitja Lenic obtained his BS, MS, and PhD in computer science from the University of Maribor Slovenia. He joined the Department of Computer Science at the University of Maribor as an assistant researcher in 1998, where he is now a teaching assistant and member of the System Software Laboratory. He has written more than 100 technical research papers published in recognized international journals and major conferences, and co-authored some textbooks. His main research interests are intelligent systems, programming languages, complex systems, and chaos theory, and software quality and metrics.

Chih-Wei Lu is currently an associate professor in the Department of Information Management, and also the chief of the Computer Center, Hsiuping Institute of Technology, Taiwan. He received his BS degree in information science from Tunghai University in 1987, his MS degree in computer science from University of Southern California in 1992, and his PhD degree in information engineering and computer science from Feng Chia University in 2003. His areas of research interest include component-based software engineering, design pattern, software reuse, and software maintenance.

Steve McRobb has worked as a senior lecturer in the School of Computing, De Montfort University since 1992. Currently based in the Division of Information Management, he teaches systems analysis and design, e-business strategy and management and related subjects. He is co-author of a successful textbook on object-oriented analysis and design using UML, research associate in the School's Centre for Computing and Social Responsibility and a doctoral research supervisor with the Software Technology Research Laboratory. Before joining De Montfort, he followed a successful career in local government administration, working for Leicester City, Leicestershire County and the Yorkshire Dales National Park.

Richard Millham received his BA(Honours) from the University of Saskatchewan in Saskatoon, Canada (1992), his MSc in software engineering from the University of Abertay in Dundee, Scotland (1995), and is completing his PhD at DeMontfort University in Leicester, UK. His PhD focuses on program transformations, WSL, and UML diagram extraction from the system code of batch-oriented legacy systems. His interests lie in software evolution, legacy system migration, UML, reverse engineering, program transformations, and WSL. He has many years of IT industry experience.

Donald Needham is an associate professor of computer science at the United States Naval Academy, with research interests of software engineering and software reuse frameworks. He has been funded through several research grants including high energy laser

modeling and simulation framework evaluation (Joint Technology Office), development of a reusability analysis framework for shipbuilding components modeled in XML and Java (Electric Boat Corporation), CORBA-based access to geospatial information (Naval Research Laboratory), and work on a formal basis for propagation modeling in multi-disciplinary design optimization (NASA).

David Nutter is studying the role of awareness in distributed software engineering, with a view to improving the efficacy of distributed software engineering teams by reducing conflict between participants and providing a historical record of the interaction by engineers with the artefacts they create. Previously, he has worked on the GENESIS project (an IST-funded project to develop and release an open source software engineering environment capable of enacting processes, managing human resources, and storing artefacts such as source code and documentation). The open source software produced is available on Source Forge. Currently, he is helping to complete the Collaborative Determination, Evolution, and Evaluation of Design Spaces (CoDEEDS) project, an EPSRC-funded project which applies a collaborative design tool developed for the steel industry to the domain of software engineering.

YI-Chun Peng received his BS and MS degrees in computer science and information engineering from Tunghai University, Taichung, Taiwan, in 2000 and 2002, respectively. His research interests include software reuse, software model and API design, design pattern, and software maintenance.

Chris Phillips is an associate professor of computer science. He has been researching in the areas of human-computer interaction (HCI), software engineering (SE), and object-oriented design for the past 15 years. He is convenor of the Massey University (New Zealand) HCI Research Group. He has refereed for a number of HCI and SE journals and conferences, and has been a member of the programme committee for several international HCI conferences. He is a member of the British Computer Society, and in 1990 was granted chartered engineer status by the Engineering Council in London.

Claudia Pons is professor of logic and formal specification at the University of La Plata, Argentina. She obtained a PhD in the application of formal methods to object-oriented modeling in 1999. She has participated in several research projects and has published papers in international conferences' proceedings and journals. She co-leads a research group on formal methods in software engineering at the Lifia (Laboratorio de Investigacion y Formacion en Informatica), at the University of La Plata, Argentina. She works part-time as a trainer and consultant in object-based development with Lifia. To contact: cpons@info.unlp.edu.ar http://portal-lifia.info.unlp.edu.ar/~cpons

Ivan Porres received his MSc in computer science in 1997 at the Polytechnic University of Valencia, Spain, and in 2001, his PhD in computer science at Åbo Akademi University in Turku, Finland. His thesis, "Modeling and Analyzing Behavior in UML," studies how

to use different formal methods to ensure the correctness of software modeled using UML. Currently, he works as an assistant professor at the Department of Computer Science at Åbo Akademi University where he studies how to use models and modeling in all aspects of software engineering.

Jianjun Pu is a graduate of Beijing University of Aeronautics and Astronauts in Beijing, China (1900), earning a BSc. He is currently a PHD student in the Software Technology Research Laboratory at De Montfort University in Leicester, UK. His research interests include reverse engineering and UML. His PhD project is through wide spectrum language to acquire the importance of UML diagrams in modeling legacy COBOL systems, describe the relationship between UML diagrams, and find out the method to produce UML use case diagrams from legacy COBOL system through other diagrams. He has been working for 14 years on software test and software reengineering as a senior engineer.

Stephen Rank has a PhD in software evolution and a BSc in computer science from the University of Durham. He has worked on the RELEASE project. producing tools for source code analysis, and the GENESIS project, producing a component of a large process-centred software engineering environment. He is currently a research assistant at the University of Lincoln, investigating support for software engineering and other collaborative activities. His research interests include software architecture, evolution of software system, and collaborative software engineering.

Andreas Rausch is heading the software architecture research group at the Technische Universität Kaiserslautern, Germany. He received a PhD in 2001 from the Technische Universität München at the chair of Prof. Dr. Manfred Broy, with the dissertation titled "Componentware — Evolution-Based Development of Software Architectures." He was responsible for the research project WEIT, developing the standard system development process model of the German government and military called V-Modell XT. Andreas Rausch has joined and headed various large interdisciplinary research projects concerning the foundations of software engineering. He has been leading various industrial software projects, developing large distributed systems, and is one of the four founders of the software house 4Soft GmbH.

Chris Scogings is a senior lecturer in computer science at the Auckland campus of Massey University, New Zealand. He is interested in user interface design and software engineering, and the development of software applications to assist practitioners and researchers in these areas.

Pasha Shabalin works for the Software & Systems Engineering chair at TU Munich (Germany) in the area of formal methods in information security, and is the author of several publications in this area. He has broad experience with IT projects in the industry, including systems with special requirements on data security, and actively provides related consulting services.

François Terrier is a doctor in electronics and head of the Software for Process Safety Laboratory at the CEA-List, France. He obtained the title of professor at the National Institute of the Nuclear Sciences and Technologies where he provides courses on complex and embedded system modeling and development. The LSP Laboratory team studies and realizes methods, tools, and platforms for the development of complex real-time systems including management of safety constraints, real-time model verification and test case generation, system evolution, and reuse. He has tutorials on "Real-Time and Object Technology" and organized several workshops on advanced issues in this domain. François Terrier has personally started and led the development of the ACCORD platform for object-oriented real-time design and implementation of embedded systems. He has been acting as project leader for several national and international projects in these domains. Recently, he has set up a large common research program among CEA, INRIA, and Thales: The Carroll program focussed on model-driven engineering and component-based midllware. The LSP Laboratory has been selected to be a core partner of the network of excellence ARTIST. François Terrier is the CEA representative in ARTIST, being more particularly in charge of ensuring network relations with UML and MDA standards and related works for on embedded domain.

Patrick Tessier earned his MS in computer science in 2002 from University of Lille (France). Currently, he is doing his PhD in the Software for Process Safety Laboratory at the CEA - French Atomic Energy Agency supervised by Pr. Jean-Marc Geib (LIFL) and Dr. Sébastien Gérard (CEA-List). His PhD is about the management of the variability for the design of a real-time system family in the context of a model-driven approach and dedicated to development of distributed real-time embedded systems. More precisely, the purpose is to define and implement mechanisms in order to derive in a good way the behavioral model of a system family. He also is involved in the European project Families, an ITEA project about system family management (http://www.esi.es/en/Projects/Families/).

Scott Tilley is an associate professor in the Department of Computer Sciences at the Florida Institute of Technology. He has a PhD from the University of Victoria. His research interests include software evolution, program redocumentation, and empirical studies of information technology efficacy. He is chair of the steering committee for the IEEE Web Site Evolution (WSE) series of events, and the current president of the Association for Computing Machinery's Special Interest Group on Design of Communication (ACM SIGDOC).

Don-Lin Yang received a BE degree in computer science from Feng Chia University (Taiwan) in 1973, an MS degree in applied science from the College of William and Mary in 1979, and a PhD degree in computer science from the University of Virginia in 1985. He is currently a professor and the chair with the Department of Information Engineering and Computer Science at Feng Chia University. Prior to joining the university in 1991, he was a staff programmer at IBM Santa Teresa Laboratory from 1985 to 1987 and a member of technical staff at AT&T Bell Laboratories from 1987 to 1991. His research interests include software engineering, distributed and parallel computing, and data mining. He is a member of the IEEE computer society and the ACM.

Yi Zhang is a master's student of computer science & engineering at The University of Connecticut, with research interests of reusability and refactoring for component-based systems (UML and Java).

Index

Q

R

S

T

U

V

W

X