LNCS 3297

Nuno Jardim Nunes
Bran Selic
Alberto Rodrigues da Silva
Ambrosio Toval Alvarez (Eds.)

UML Modeling Languages and Applications

«UML» 2004 Satellite Activities
Lisbon, Portugal, October 2004
Revised Selected Papers

Springer

Nuno Jardim Nunes   Bran Selic
Alberto Rodrigues da Silva
Ambrosio Toval Alvarez (Eds.)

UML Modeling Languages and Applications

«UML» 2004 Satellite Activities
Lisbon, Portugal, October 11-15, 2004
Revised Selected Papers

Springer

Volume Editors

Nuno Jardim Nunes
Universidade da Madeira, Campus da Penteada
Mathematics and Engineering Department
9000-390 Funchal, Portugal
E-mail: njn@uma.pt

Bran Selic
IBM Rational Software
770 Palladium Drive, Kanata, Ontario K2V 1C8, Canada
E-mail: bselic@ca.ibm.com

Alberto Rodrigues da Silva
INESC-ID and Instituto Superior Técnico
Rua Alves Redol, 9, 1000-029 Lisboa, Portugal
E-mail: alberto.silva@acm.org

Ambrosio Toval Alvarez
Universidad de Murcia, Campus de Espinardo
30100 Murcia, Spain
E-mail: atoval@um.es

Library of Congress Control Number: 2005921517

CR Subject Classification (1998): D.2, D.3, K.6, I.6

ISSN 0302-9743
ISBN 3-540-25081-6 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11399971 06/3142 5 4 3 2 1 0

Preface

The ≪UML≫ 2004 conference was held in Lisbon (Portugal) from October 11 through October 15, 2004. It was the seventh conference in a series of annual events that started in 1998. ≪UML≫ has rapidly become one of the leading venues to present and discuss the development of object-oriented modeling. In order to reflect the changes in the field, the ≪UML≫ conference series will be continued from 2005 onwards under the name MODELS (Model Driven Engineering, Languages and Systems).

In an effort to make this year's conference more useful and effective for a wider community, including academics and practitioners working in areas related to UML and modeling in general, a set of satellite events was organized, including workshops dedicated to specific research topics, an industry track, a poster/demo session, and a tools exhibit. This volume is a compilation of the contributions presented at these satellite events.

Workshops at ≪UML≫ 2004 took place during the first three days of the conference (from October 10 to 12). Following the tradition of previous ≪UML≫ conferences, ≪UML≫ 2004 workshops provided a collaborative forum for groups of (typically 15 to 30) participants to exchange recent or preliminary results, to conduct intensive discussions on a particular topic, or to coordinate efforts between representatives of a technical community. Ten workshops were held, covering a variety of hot topics, which have been covered in the workshop reports contained in this volume. Each workshop lasted for a full day. A novelty with respect to previous ≪UML≫ conferences was the inclusion of a Doctoral Symposium, which was well received, to provide an explicit space for young researchers developing their thesis on some aspect related to UML. We would like to emphasize the relevant and innovative topics considered in the workshops at this ≪UML≫ conference edition as well as the high level of participation in all of them. All these workshops were selected by the Program Committee indicated below, after a formal review process. Special thanks are given to all members for their valuable support. We would like to thank also the Spanish Ministry of Science and Technology (project DYNAMICA/PRESSURE TIC 2003-07804-C05-05) for its aid in the workshop reports' publication.

The purpose of the industry track was to report on innovative results and experiences in the industrial application of software modeling and model-driven development in industrial settings. Competitive pressures and the all too familiar problems of traditional programming-oriented approaches to software development have led to some remarkable and highly innovative applications of model-driven development in industry. The organizing committee of the ≪UML≫ conference series felt that it was appropriate and timely to provide a forum where such results could be reported, not only to describe new and interesting techniques and technologies but also to demonstrate that model-driven development has been applied successfully and widely beyond the research lab environment.

The resulting 12 industry papers—all peer reviewed—combine a set of submissions selected by a dedicated Industry Track Program Committee (see below) and a set of invited papers from domain experts with proven results in industry. These papers are not mere experience reports, although experience is an important facet of all of them, but they describe important contributions to the evolving body of theory of model-driven development.

The poster/demo session took place during the main conference, from October 13 to 15. The 11 accepted submissions were displayed in the coffee-break room, enabling contributors to get the most feedback on their work. The list of accepted posters/demos spans many different research areas, from model-based testing to user-interface design; and also many application domains, from distributed systems to critical systems. The poster/demo contributions are available in this volume in the form of extended short papers. One of the goals of the UML is to provide tool support and interchange. Here you have an opportunity to be familiar with many interesting research projects.

Live demonstrations of cutting-edge systems were an important and exciting part of the conference. The tool exhibits session provided an excellent opportunity where participants analyzed and viewed the most relevant UML- and MDA-related tools in action and discussed these systems with their creators or distributors. The tool exhibits session took place during the main conference, from October 13 to 15, and included the following live demos: (1) "seCAKE: A Complete CASE Tool with Reuse Support", dTinf; (2) "Making UML Diagrams Accessible for Visually Impaired Programmers", FNB; (3) "The Suite of Telelogic Products: Doors/Analyst, TAU/Developer, and TAU/Architect", Telelogic; (4) "IBM Rational Rose XDE Products", Sinfic; and (5) "Nucleus BridgePoint", Mentor Graphics. The tool exhibit contributions are available in this volume in the form of extended short papers.

Following this preface are several important pages listing the many people and organizations without which this event would not have been possible. Please take a moment to peruse these pages and join us in thanking them for their dedication and support. We also thank the staff at Springer for the help with the production of this volume.

Finally we think the excellent contributions in this volume speak for themselves.

November 2004

Nuno Jardim Nunes
Bran Selic
Alberto Silva
Ambrosio Toval

Organization

Executive Committee

General Chair	Stephen J. Mellor (Mentor Graphics, USA)
Conference Chair	Ana Moreira (New University of Lisbon, Portugal)
Program Co-chairs	Thomas Baar (EPFL, Switzerland)
	Alfred Strohmeier (EPFL, Switzerland)
Industry Track Chair	Bran Selic (IBM Rational Software, Canada)
Tutorials Chair	Ezra K. Mugisa (University of the West Indies at Mona, Jamaica)
Workshop Chair	Ambrosio Toval (University of Murcia, Spain)
Panel Chair	Jon Whittle (NASA Ames Research Center, USA)
Posters Chair	Nuno Jardim Nunes (University of Madeira, Portugal)

Organizing Team

Publicity Chairs	João Araújo (New University of Lisbon, Portugal)
	Geri Georg (Colorado State University, USA)
Local Arrangements Chair	Isabel Sofia Brito (Politécnico de Beja, Portugal)
Tools Exhibition Chair	Alberto Silva (Technical University of Lisbon, Portugal)
Local Sponsors Chair	Fernando Brito e Abreu (New University of Lisbon, Portugal)
Web Chair	Miguel Goulão (New University of Lisbon, Portugal)

Workshop Program Committee

Eric Dubois (Luxembourg)
Jean Michel Bruel (France)
Juan Hernández (Spain)
Ivan Porres (Finland)
Roel Wieringa (The Netherlands)

Industry Track Program Committee

Posters/Demos Program Committee

Sponsors

SINFIC
http://www.sinfic.pt

Springer Verlag
http://www.springeronline.com

Mentor Graphics
http://www.mentor.com

IBM France
http://www.ibm.com/fr

Supporters

ACM Special Interest Group
on Software Engineering
http://www.acm.org

IEEE Computer Society
http://www.ieee.com

New University of Lisbon
http://di.fct.unl.pt

Turismo de Lisboa
http://www.tourismlisbon.com

Object Management Group,
http://www.omg.org

Table of Contents

Consistency Problems in UML-Based Software Development

Zbigniew Huzar[1], Ludwik Kuzniarz[2], Gianna Reggio[3], and Jean Louis Sourrouille[4]

[1] Department of Computer Science, Worcław University of Technology, Worcław, Poland
[2] School of Engineering, Blekinge Institute of Technology, Ronneby, Sweden
[3] DISI, Università di Genova, Genova, Italy
[4] Department of Information Technology and Computer Engineering, INSA, Lyon, France

Abstract. This survey of the workshop series *Consistency Problems in UML-based Software Development* aims to help readers to find the guidelines of the papers. First, general considerations about consistency and related problems are discussed. Next, the approaches proposed in the workshop papers to handle the problems are categorized and summarized. The last section includes extended abstracts of the papers from the current workshop.

1 Why Consistency?

The introduction of the first workshop could have been the same for the series: *The Unified Modeling Language (UML) has become an industrially accepted standard for object-oriented modeling of large, complex systems as well as a basis for software development methodologies. During the development process, artifacts representing different aspects of the system are produced. The artifacts should be properly related to each other in order to form a consistent description of the developed system. The problems concerning and related to consistency between diagrams and models produced within the UML-based development process are presented and discussed within the scope of the workshop. In particular, two kinds of problems concerning consistency are addressed – those related to consistency between diagrams within a given model and named as an intra-consistency problem and those concerning consistency between different models and named as an inter-consistency problem. The papers selected and included in the workshop materials are intended to present a spectrum of problems that occur when consistency is concerned, starting from a general perspective and methodology for systematic checking of consistency, through possible ways of extending UML to enable consistency checking and checking consistency through model transformations, followed by examples of practical realization of the checking in practice and possible tools support, ending with formalization of the notions of consistency.*

The number of submissions and participants shows the importance of the issue. Each workshop proposed to focus on particular topics: *consistency definition and* *verification* (I), *examples of inconsistencies* (II), and *dependency relationship* (III). However, the papers tackle problems in all areas related to consistency, from several points of view and using various approaches.

1.1 Intra-model Consistency

Consistency problems do not seem to arise in many notations such as programming languages. So, a preliminary question is: Where are the UML consistency problems coming from?

When using the UML during the development process, many artifacts representing different aspects of the system are produced, and these artifacts should be properly related to each other in order to form a consistent description of the developed system. There are two main reasons for having many different UML artifacts describing the same system:

- multiview nature of UML models: at some level of abstraction a system is described as a collection of views dealing with different, possibly overlapping, aspects,
- the system is developed throughout different phases and iterations, with each one producing a new, more refined description of the system.

Another source of inconsistency is the imprecise semantics of the UML. A UML expression (i.e., a set of model elements) may have multiple interpretations, among which some are inconsistent. Why is the UML semantics not precise? It was the wish aim of the UML authors not to supply a precise definition of the UML to broaden the area in which the UML applies. An advantage is that such imprecise UML models can be implemented in many ways. The counterpart is that we do not know if there is one possible implementation of a UML model. This issue is called intra-model or horizontal consistency. For instance, intra-consistency is expected between model elements representing the static and dynamic views of the modeled domain.

1.2 Inter-model Consistency

Furthermore, consistency problems arise in the UML because there is no definition of relationships between models preserving consistency such as the refinement relationship. A UML–based software development is a modeling process. From the requirements to the code, the software development process produces more and more detailed models. A model is a collection of UML model elements that represent a system at a given level of abstraction. At each level, the produced model should be consistent with the models at the upper, more abstract levels. This issue is called inter-model or vertical consistency. For example, a design model should be inter-consistent with an analysis model.

1.3 Main Issues Related to Consistency Addressed in Contributions

The papers presented and discussed during the workshops deal with the following important issues: definition of consistency, relationships between consistency and development process, approaches to check consistency, and checking tools. The positions are briefly summarized below. Regarding tools, the two main approaches are:

- to check directly that the UML model has the required properties (expressed by OCL or other means), using standard tools when available, and
- to translate the model into a formal language such as B or production rules, and then to perform checks using companion tools of the target language.

2 A Survey of the Workshop Contributions

2.1 Consistency Definitions

Rule-Based Definitions

The semantics of the UML includes constraints that induce restrictions on the use of notations in order to ensure that model interpretations are licit [26]. To avoid inconsistencies and to make the semantics precise, most papers propose adding constraints or well-formedness rules such as the UML ones [4,7,8,9,11,19,21,25, 26,27,30]. A model is inconsistent when it violates the added constraints, i.e., when there is no licit interpretation [27]. In [20], a class diagram is consistent if there is at least one instantiation possible that satisfies all the diagram constraints. UML artifacts form a hierarchy and all the components of an artifact should be intra and inter consistent for the artifact to be consistent [11]. Some papers only deal with model properties that do not ensure the entire model consistency: the behavior should be deadlock free [24], sequence diagrams should be consistent with statecharts [3,15], etc.

[2] presents an approach to define which UML models are intra-consistent following an algebraic approach, that is distinguishing in a UML model a "signature" which defines the model vocabulary, which is then used to check the well-definedness of the other parts in quite a modular way.

Refinement

Furthermore, constraints are added to enforce the inter-model consistency, i.e., to define the refinement relationship. Applying the ODP consistency approach [6], a set of specifications (models) is consistent if there exists a specification that is the refinement of each of the specifications in the set with respect to a refinement relationship. In [21], consistency constraints include conformance to standard, good practice and stakeholders´ specific constraints. [13] presents a general framework for defining refinement relationships between UML models, trying to distinguish between abstraction refinement and semantics refinement, where only the first may be automatically checked.

Translational Definitions

Adding constraints can be seen as a declarative approach. In a translational approach, a model is consistent if its translation into a formal language (such as B or Object-Z) satisfies some good properties [6,24,23,20,3]. This approach does not enforce the entire UML model consistency, for instance in [20] only class diagrams, object diagrams and statecharts are taken into account, while [24] only deals with behavior. Quite different, but also based on transformations, graphical consistency conditions specify the situations that must not occur [16]. [22] introduces a formal language OOL, and proposes transforming a subset of UML models into OOL specifications. The well-defined consistency and a refinement calculus of OOL are then used to check the corresponding UML models.

Constraint Completeness
A further question is to write the entire list of constraints: examples of classification are given (between pairs of diagrams in [8], by abstraction levels in [26]), but it is likely that no complete list exists. Assuming that syntactic rules are expressed formally and semantic rules use natural language, if all the constraints cannot be expressed by syntactic rules, consistency cannot be checked automatically [21,27].

Role of Dependency in Defining Consistency
[29] presents a UML profile allowing one to express dependency relationships among model elements characterized by behavioral properties, such as *call/update/access* preservation, to help establish correct refinement among models. These relationships are formally defined using Description Logic. Similarly, [17] sketches another profile for expressing different kinds of dependency, precisely implicit and explicit usage among model elements.

2.2 Consistency and Development Process

Refinement
During the development process, model consistency should be preserved through refinement: Object-Z and CSP provide refinement concepts for checking the translations of UML models in [24], while in [16] model transformations are expressed using graphs. Another approach proposes defining a profile with transformation rules using the UML extension mechanisms [25].

Development Methodology
Moreover, models should be consistent with the development methodology or process (e.g., USDP: Unified Software Development Process in [11], COMET in [8], general process in [19]). In [11] a three-layer framework is adapted to the development process, while [6] uses the ODP principle of viewpoints (i.e., partial specifications) to check UML consistency. Good practice rules and specific development rules should also be added [21] or followed [18], preferably in a UML profile [25,27]. [5], instead, considers the consistency problem in the component-based development process *KobrA*. [14] considers the problem of the consistency among the artifacts produced following the USDP and proposes a UML profile expressing such artifacts and defining rules expressed with OCL to enforce

consistency; such rules are then checked using any standard OCL tool. [2] proposes a UML based development method which requires models to be produced with a precise structure, and equipped with guidelines helping to detect the most probable inconsistencies.

Incompleteness
Several authors underline that the under-specification of the UML induces incompleteness [26,19], while models should be complete for consistency checks. Rules can be checked on existing models, and examples of results given in [19] show that inconsistencies are related to the development practices of the designers.

Domain Specific Cases
[10] presents the consistency aspects of the MERODE method for developing information systems; the method is based on the formal language CSP and proposes the use of views of three different kinds, with two having a UML-like syntax. [1] treats consistencies due to a too rigid application of design patterns; to avoid these they propose presenting patterns using an extension of the UML 2.0 collaboration template, which allows to constrain the parameters and to perform some actions at the instantiation time, such as deletion of model elements.

2.3 Consistency Checking

Most papers deal with either intra-model consistency [4,6,7,8,9,11,15,20,21,23,24, 26], some with inter-model consistency [11,16,25], others deal with both, such as [23], which translates models into B that supplies a refinement relationship. Obviously, a tool is required to check consistency, not only to automatically check constraints but also to help users to find and correct errors. Depending on the approach, declarative or translational, tools are faced with different problems. Examples of checks applied to models are given in [14,15,18].

Constraint Checking
Each tool is associated with a suitable representation for the constraints. The most direct way to express constraints is the OCL (Object Constraint Language). The checking tool is standard and could be embedded in the modeling tools, as in [14]. The OCL used to express the rules is enriched with a transitive closure operator and temporal operators in [4], and with actions in [8]. [26,21] use production rules that add reasoning capabilities to constraints, and unlike OCL which is side-effect free, allow actions such as corrections or tips. The graph rewriting rules in [30] describe the resolution actions for detected inconsistencies. Based on rules in XML, the *xlinkit* framework allows checking consistency of models mapped to XML using the XMI [9]. The graphical conditions in [16] are kinds of patterns, and checking constraints comes down to matching graphs in the UML model. [29] describes a tool, RACOoN, for checking consistency conditions expressed in Description Logic by combining a UML tool, an XML translator and a logical reasoning tool.

Model Translation
Model translation into a formal language is very appealing since checking tools already exist. Only the notions common to UML and the target language can be translated, and the inter-consistency definition depends on its refinement relationship. In [6] a detailed discussion of the translational approaches illustrated with LOTOS and Object-Z is given. In [24], static aspects are translated into Object-Z while behavioral ones are translated into CSP: only deadlocks and interface properties are checked. The B specification and the UML model are handled in parallel in [7], but the question of how to automate the translation of UML/OCL models into B is not answered. UML models are decorated with additional expressions to allow the translation into B in [20], but for temporal properties another approach is proposed. LTS and traces are used in [3] to check behavior properties. [5] proposes to reduce the consistency issues into deadlock detection problems to be checked using the SPIN tools. On the other hand, [10] describes a tool, MERMAID, which monitors the constructions of the models required by the MERODE method, helping to ensure their consistency (also if some post-mortem checks are implemented). Simulation approaches do not give proofs but they increase the confidence in the model, e.g., trace validation in [15]. Translation to description logic is suggested in [28] to maintain consistency, together with the use of an accompanying tool to prove the feasibility of the approach. Consistency checking based on the consistency rules expressed as graphs rewriting rules and their implementation in the UML CASE tool is presented in [30].

3 Extended Abstracts

On Understanding of Refinement Relationship
Bogumiła Hnatkowska, Zbigniew Huzar, Lech Tuzinkiewicz
The software development process is both iterative and incremental in nature. Modeling constitutes an important step of this process; its key artifacts are described as models, i.e. abstract representations of the entities being modeled. There are many relationships between models. The «refine» relationship is an interesting one as it reflects the evolution of artifacts within the software development process. The relationship is not precisely defined in the UML standard. Its informal definition relates to other, not well-defined notions: “perspective”, “abstraction level”, and “semantic level”. The paper proposes definitions of these notions in the UML terms. Refinements defined in the paper are based on the change of abstraction levels and on the change of semantic levels. The first kind of refinement is independent of the interpretation of models while the second kind depends on model interpretation. Therefore two models’ categories were introduced: non-interpretable and interpretable, based on the formal definition of abstract and semantic levels. The elaborated definitions may be used for describing different step-wise model transformations.

Consistency and Refinement of UML Models

Zhiming Liu, He Jifeng, Xiaoshan Li, Yifeng Chen

In UML-based software development, artifacts created in the development process are modeled and analyzed from static and dynamic views using different kinds of UML notations. Under the multiple views of UML, the developers can decompose a software design into smaller parts of manageable scales. A development process starts from a system requirement model consisting of a class diagram, a family of sequence diagrams, and a family of state diagrams. Such a model can be established through horizontal requirement incrementally by adding information and incorporating use cases one by one. A development process also cycles through a number of steps of vertical refinement from the requirement model into a system design model. Therefore, the horizontal and vertical consistency are the inevitable challenging issues, which arise from such a multi-view and multi-notational approach.

In this paper, we use a formal object-oriented specification language (OOL) to formalize and combine UML models. With OOL, a specification of an object system is a combination of its class declarations, method declarations and specifications of method bodies. Different sub-models of a system model are formalized as different parts in an OOL specification. The consistency of the different sub-models is defined as the well-formedness of the corresponding OOL specification. With the formalization, we develop a set of refinement laws of UML models to capture the essential nature, principles, nature and patterns of object-oriented design. We can apply the refinement calculus of OOL specifications to treat refinement of system models in UML. With the support of the incremental and iterative features of object-orientation and the Rational Unified Process (RUP), the refinement process will preserve the consistency and correctness of the system.

UML 2.0 Model Consistency – The Rule of Explicit and Implicit Usage Dependencies

Shiri Kremer-Davidson, Yael Shaham-Gafni

The notion of dependency is modeled in UML using the Dependency relation. The UML specification intentionally defines the Dependency concept vaguely in order to serve as a "catch all" relation, describing any relationship that is not a generalization or association. The specification further defines several subtypes of Dependency: Abstraction, Realization, Substitution, and Usage, which have a stronger semantic meaning. For all of these modeling constructs the UML specification does not describe any relation to the behavioral aspects or to model elements representing runtime entities.

In this paper we investigate the runtime implications for the usage dependency. We define the notions of explicit dependencies: dependencies that are explicitly created by the modeler as part of the static aspect of a UML model, and implicit usage dependencies: usages that can be inferred form the behavioral portions of a UML model. Based on these notions we propose a definition for the semantics of the usage dependency and a corresponding consistency notion. We propose an implementation of such semantics and consistency through a UML Profile. We provide an example to illuminate our ideas and describe several scenarios in which having knowledge of the explicit and implicit dependency information and the consistency between them is beneficial.

Consistency Checking of USDP Models

Bogumila Hnatkowska, Anita Walkowiak

The aim of the paper is to propose a method for checking consistency of UML models. Because the content of UML models strongly depends on used methodology it was assumed that models that are basic outcomes of USDP process are considered. Our aim was to improve the USDP process with some mechanisms validating prepared models against some known rules. The rules belong to two categories:

– well-formed rules, defined in UML standard document,
– new well-formed rules resulting from applying USDP for software development.

In the paper three USDP models are refined and formalized, i.e. *Context Model*, *Use Case Model*, and *Analysis Model*. The models are defined in terms of a new language called *Robust Software Development UML* (*RSD_UML*). *RSD_UML* is a part of *Robust Software Development Profile* (*RSDP*). The profile introduces new stereotypes basing on standard UML elements. *RSD_UML* language is defined similarly to UML standard. Its syntax and static semantics are defined formally by OCL expressions, while its dynamic semantics is defined informally, in natural language. It was observed that most of the intra-consistency rules relate to the way of proper construction of models. For example, the rules state that collaboration at given semantic level (e.g. analysis) should represent a behavioral element from the previous model (e.g. requirements). Example models written in XMI were prepared in two different CASE tools, i.e. Rational Rose, and Poseidon for UML. OCL Evaluator was used for models verification against inter and intra-consistency rules.

Formalizing Behaviour Preserving Dependencies in UML

Ragnhild Van Der Straeten

In the context of Model-Driven Development (MDD), models are primary assets that embody a consistent view on the system under study. On the one hand, during model driven software development, software models can evolve into a new version.

Model refactorings are a particular kind of model evolution which preserve the behaviour as specified by the model. On the other hand, within the software development life-cycle, models can gradually be refined resulting in a full-fledged implementation. At every refinement step, this refinement process adds more concrete details to the model. In general, refinements preserve certain correctness issues, e.g. program refinements imply the preservation of program correctness. The behaviour preserving properties identified in this paper for model refactorings can also be used in the context of refinements. These properties express that certain parts of the specified behaviour have to be preserved. In the context of model refinements, these behaviour preserving properties can be interpreted as correctness properties between a certain model and its refined version. In the rest of the paper, we refer to these properties as behaviour preserving properties. The goal of this paper is threefold. First of all, definitions of behaviour preserving properties are given in the context of UML models. During the development process, we also want to indicate between which UML elements or models certain properties are valid. In UML the dependency relationship is used to describe relationships between models and their elements.

However, it lacks a precise definition. Thus, the second aim of the paper is to extend the UML metamodel with specialized dependency relationships expressing the preservation properties. Thirdly, these dependency relationships are formalized using a logic approach. This allows the automatic checking of these relationships between UML models and elements. This is illustrated by a simple but nevertheless representative example.

Behavioral Consistency Checking for Component-Based Software Development Using the KobrA Approach
Yunja Choi, Christian Bunse
The KobrA method is a structured approach for component-based system development, providing a natural way of identifying and refining system components by separating the external view (interface or contract) from its internal view (detailed functionalities and their realization). The method is designed to reduce system complexity by separating concerns and facilitates software reuse, thus, saving time and effort for software development.

Nevertheless, understanding the overall interactions and relations of many components in a KobrA model often goes beyond human capability, mainly due to its way of specifying different aspects of a component in various UML diagrams. For example, statecharts are used to specify the abstract level component behavior and activity/sequence/collaboration diagrams are used to specify detailed internal component behavior. While this approach facilitates a systematic, iterative specification-refinement paradigm, it can also produce unexpected inconsistencies among these different diagrams as well as among the different levels of refinement. A systematic consistency checking mechanism is a must to ensure the basic quality of a system.

In this paper, we aim at providing an overall consistency checking mechanism integrated into the development process of KobrA, named *consistency checking using environment modeling*. We first define generic consistency requirements in the KobrA approach, with an emphasis on the behavioral consistency between different levels of specifications and realizations. The consistency requirements are then reinterpreted as consistency between a set of state transition systems describing the system behavior (*reactive* systems) and a sequence of stimuli describing the system environment (*action* systems). Two behavioral models are considered consistent if the reactive system accepts every stimulus generated by the action system. In this way, we transform various consistency issues into a deadlock detection problem that can be automated. We demonstrate the automated consistency checking using the model checker SPIN on a hypothetical elevator system.

Implementing Consistency Management Techniques for Conceptual Modeling
Raf Haesen, Monique Snoeck
Most software development methodologies justify the use of multiple independent models to represent all aspects at the different stages in the development process. This can make the resulting information system inconsistent at different levels: inconsistencies can arise between different views of a single system, between

documents at different development life cycle stages, or in a single document. The use of a single model and different views to that model can avoid this problem: all views have to be built according to well-formedness rules for that view and consistency between the related views must be checked. In this way it is possible to obtain a model that reaches a feasible level of validity and improved completeness. Validity means that all statements made by the model are correct and relevant to the problem, whereas completeness means that the model contains all the statements about the domain. This paper presents different techniques to maintain consistency of one view and the use of the same techniques to enforce and check consistency between the views. First we discuss the three strategies of consistency management: consistency by analysis, consistency by monitoring and consistency by construction. Finally we present a concrete implementation of these rules in a modeling tool, based on the object-oriented domain modeling method MERODE.

Improving Pattern Support in UML CASE Tool
Samir Ammour, Xavier Blanc, Mikal Ziane, Philippe Desfray
In this paper we improve the UML2.0 Collaboration Templates mechanism to better support patterns in UML CASE tools. In our research and prototyping activities, we have identified that two important problems lead to severe limitations: Collaboration Templates are not versatile enough to support design patterns correctly. First, they constrain their parameters inappropriately. Second, the instantiation of UML Collaboration Templates does not allow us to modify or to suppress model elements, which is sometimes necessary. Both problems make it difficult to maintain the UML models' consistency when applying design patterns. Collaboration Templates may lead to inconsistencies in models. We thus propose to explicitly constrain Collaboration Template parameters using pattern constraints and to allow the suppression or modification of model elements using pattern actions. Pattern constraints are OCL expressions to control which elements can be bound to the template parameters to preserve the consistency of models. Pattern actions are written in an action language such as Action Semantics or an extension of OCL. They are used to modify and delete model elements to remove inconsistencies when applying design patterns. We have prototyped this approach in the Objecteering UML CASE tool. Both these improvements proved quite useful in several applications, and will be included in a future version of the Objecteering CASE tool.

References

I. L. Kuzniarz, G. Reggio, J.L. Sourrouille, Z. Huzar, *Workshop on Consistency Problems in UML-based Software Development I*, UML 2002, Blekinge Institute of Technology, Research Report 2002:06. Available at `http://www.ipd.bth.se/uml2002/`.
II. L. Kuzniarz, G. Reggio, J.L. Sourrouille, Z. Huzar, M. Staron, *Workshop on Consistency Problems in UML-based Software Development II*, UML 2003, Blekinge Institute of Technology. Research Report 2003:06. Available at `http://www.ipd.bth.se/consistencyUML/UML2003workshop.asp`.

III. Z. Huzar, L. Kuzniarz, G. Reggio, J.L. Sourrouille, *Workshop on Consistency Problems in UML-based Software Development III*, UML 2004. Available at `http://uml04.ci.pwr.wroc.pl/`.

1. S. Ammour, X. Blanc, M. Ziane and P. Desfray, *Improving Pattern Support in UML CASE Tools,* in III
2. E. Astesiano and G. Reggio, *An Algebraic Proposal for Handling UML Consistency,* in II
3. P. Bhaduri and R. Venkatesh, *Formal Consistency of Models in Multi-View Modelling,* in I
4. J.-P. Bodeveix, T. Millan, C. Percebois, C. Le Camus, P. Bazex and L. Feraud, *Extending OCL for verifying UML models consistency,* in I
5. Y. Choi and C. Bunse, *Behavioral Consistency Checking for Component-based Software Development Using the KobrA Approach,* in III
6. J. Derrick, D. Akehurst and E. Boiten, *A framework for UML consistency,* in I
7. G. Génova, J. Llorens and J. M. Fuentes, *The Baseless Links Problem,* in II
8. H. Gomaa and D. Wijesekera, *Consistency in Multiple-View UML Models: A Case Study,* in II
9. C. Gryce, A. Finkelstein and C. Nentwitch, *Lightweight Checking for UML Based Software Development,* in I
10. R. Haesen and M. Snoeck, *Implementing Consistency Management Techniques for Conceptual Modeling,* in III
11. B. Hnatkowska, Z. Huzar, L Kurniarz and L. Tuzinkiewicz, *A systematic approach to consistency within UML based software development process,* in I
12. B. Hnatkowska, Z. Huzar, L Kuzniarz and L. Tuzinkiewicz, *Refinement relationship between collaborations,* in II
13. B. Hnatkowska, Z. Huzar and L. Tuzinkiewicz, *On Understanding of Refinement Relationship,* in III
14. B. Hnatkowska and A. Walkowiak, *Consistency Checking of USDP Models,* in III
15. T. Huining Feng and H. Vangheluwe, *Case Study: Consistency Problems in a UML Model of a Chat Room,* in II
16. J. Hendrik Kausmann, R. Heckel and S. Sauer, *Extended Model Relations with Graphical Consistency Conditions,* in I
17. S. Kremer-Davidson and Y. Shaham-Gafni, *UML 2.0 Model Consistency – the Rule of Explicit and Implicit Usage Dependencies,* in III
18. L. Kuzniarz and M. Staron, *Inconsistencies in Student Designs,* in II
19. C. Lange, M.R.V. Chaudron, J. Muskens, L.J. Somers and H.M. Dortmans, *An Empirical Investigation in Quantifying Inconsistency and Incompleteness of UML Designs,* in II
20. K. Lano, D. Clark and K. Androutsopoulos, *Formalising Inter-model Consistency for the UML,* in I
21. W. Qian Liu, S. Easterbrook and J. Mylopoulos, *Rule Based detection of Inconsistency in UML Models,* in I
22. Z. Liu, H. Jifeng, X. Li and Y. Chen*, Consistency and Refinement of UML Models,* in III
23. R. Marcano and N. Levy, *Using B formal specifications for analysis and verification of UML/OCL models,* in I
24. H. Rasch and H. Werheim, *Consistency between UML Classes and Associated State Machines,* in I
25. W. Shen, Y. Lu and W. Liong Low, *Extending the UML Metamodel to Support Software Refinement,* in II

26. J.-L. Sourrouille and G. Caplat, *Checking UML Model Consistency,* in I
27. J.-L. Sourrouille and G. Caplat, *A Pragmatic View on Consistency Checking of UML Models,* in II
28. R. Van Der Straeten, T. Mens and J. Simmonds, *Maintaining Consistency between UML Models Using Description Logic,* in II
29. R. Van Der Straeten, *Formalizing Behaviour Preserving Dependencies in UML,* in III
30. R. Wagner, H. Giese and U. A. Nickel, *A Plug-In for Flexible and Incremental Consistency Management,* in II

5th International Workshop on Aspect-Oriented Modeling

Dominik Stein[1], Jörg Kienzle[2], and Mohamed Kandé[3]

[1] University of Duisburg-Essen, Schützenbahn 70, D-45117 Essen, Germany
dstein@cs.uni-essen.de
[2] School of Computer Science, McGill University, Montreal, QC H3A 2A7, Canada
Joerg.Kienzle@mcgill.ca
[3] Condris Technologies, Switzerland
Mohamed.Kande@condris.com

Abstract. This report summarizes the outcome of the 5th Workshop on Aspect-Oriented Modeling (AOM) held in conjunction with the 7th International Conference on the Unified Modeling Language – UML 2004 – in Lisbon, Portugal. The workshop brought together researchers and practitioners from two communities: aspect-oriented software development (AOSD) and software model engineering. It provided a forum for discussing the state of the art in modeling crosscutting concerns at different stages of the software development process: requirements elicitation and analysis, software architecture, detailed design, and mapping to aspect-oriented programming constructs.. This paper gives an overview of the accepted submissions, and summarizes the results of the different discussion groups.

1 Introduction

This paper summarizes the outcome of the 5th edition of the successful Aspect-Oriented Modeling Workshop series. An overview of what happened at previous editions of the workshop can be found at [12].

The workshop took place at the Vila Galé Opéra Hotel in Lisbon, Portugal, on Monday, October 11th 2004, as part of the 7th International Conference on the Unified Modeling Language – UML 2004 [1]. Participation was based on the submission of a position paper addressing aspect-oriented modeling issues. A total of 17 papers were submitted and reviewed by the program committee, 13 of which were accepted to the workshop. In order to leave enough time for discussion, only a one-and-a-half-hour session was dedicated to presentations. Four papers were chosen as representatives of the workshop submissions, and intended to stimulate and provide provocative input to the following discussion sessions. In the late morning session, the attendees split into three groups to independently discuss specific questions concerning the eligibility of concrete UML model elements to represent aspect-oriented concepts. From there, the discussions quickly led to other topics. The results of the discussion groups were collected at the end of the workshop, and presented and re-discussed with the entirety of the workshop participants. Finally, a catalogue of questions indented as an agenda for future research was established.

The rest of this report is structured as follows: Section 2 gives an overview to the accepted papers. Section 3 summarizes the results of the discussion groups. Section 4 concludes the report and presents identified future research directions.

2 Overview of Accepted Position Papers

A total of 13 papers were accepted to the workshop (see below). This section presents a brief overview of the papers, organized according to the software development life cycle phase they apply to. This structure should make it easier for an interested reader to identify the submissions that pertain to his / her research area.

Some of the papers deal with modeling of aspects at the requirements phase: *Navarro et al.* (8), for example, introduce a UML profile for ATRIUM [7], a methodology that allows the concurrent definition of requirements and software architectures using both goal-oriented and aspect-oriented techniques. The profile presented in the paper comes with a graphical notation that helps visualizing goal models. Within those goal models, candidate aspects arise implicitly from goals that participate in weaving relationships. *Spies et al.* (11) explain i* [13] ("eye-star"), an early-requirements engineering technique developed by *Yu*. They describe how the models being generated using this technique may be mapped to "concern templates" as introduced by *Brito* and *Moreira* [2], thus allowing the identification of candidate aspects.

One paper concentrated on domain models: In his paper, *Steimann* (12) claims that domain models are inherently aspect free. He picks up common usages of the term "aspect" in modeling, and investigates their meaning in software modeling: He compares aspects to roles, discusses aspects as ordering dimensions, takes a look at domain-specific aspects, and reasons on aspects of modeling. Finally, he provides a proof of his claim with help of predicate logic.

Other papers focus on the impact of aspect-oriented techniques on the design phase and the design process: For example, *Li et al.* (4) elucidate the necessity of having an aspect-oriented design in order to achieve the full benefits when it comes to (aspect-oriented) programming. On the other hand, they identify limitations in prevailing aspect-oriented programming languages that impose important drawbacks on the (aspect-oriented) design. Having said that, they describe a development process that copes with these problems. *Park* and *Kang* (9) present an approach to support design phase performance simulation and analysis with help of aspect-oriented techniques. They make use of separate models to specify the core functionality and the performance requirements of a system, and map them to distinct AspectJ [6] code modules. By keeping performance specifications separate from functional specifications, they gain benefits in both the specification and the adaptation ("feedback") of the performance requirements. *Kamalakar* and *Ghosh* (3) illustrate how the aspect-oriented modeling technique developed by *France et al.* [3] can be used to encapsulate middleware-specific functionality in distinct design models. Later, these models can be woven into models specifying the business functionality of an arbitrary application, thus leading to a higher reusability of the middleware-specific design. The ideas are illustrated on a case study using CORBA.

Besides that, some papers were concerned with the verification and testing of aspect-oriented models: *Nakajima* and *Tamai* (7), for example, propose to use Alloy [5], a lightweight formal specification language and analysis tool, for the precise description of models as well as their verification. They demonstrate the use of Alloy – i.e., how to specify aspects and how to weave them – with help of two examples, logging and access control. *Tessier et al.* (13) present a formalized method, based on static model analysis, for the early detection of conflict in aspect models. To do so, they investigate the crosscutting relationships between aspects and their base classes, summarize the outcome in a table, and feed the collected data to formal expressions. That way they are able to detect possible conflicts in the ordering of aspects, in transverse specifications, or accidental recursion, etc.

Multiple papers deal with modeling notations: *Barra et al.* (1) investigate the UML 2.0 specification [10] in order to find suitable abstraction means for the representation of aspect-oriented concepts. In particular, they analyze the new UML 2.0 elements Ports and Connectors – in connection with the revised specification of Interfaces, Association Classes, and Components – and discuss their eligibility to represent join points, advice, introductions, etc. *Han et al.* (2) introduce a MOF [9] meta-model for AspectJ – capturing the syntax and semantics of each of its language constructs – in order to allow the modeling of AspectJ programs. They give a detailed description of each meta-class, its semantics, its attributes, as well as the associations it participates in. Further, they provide a corresponding visualization. *Muller* (6) presents View Components, an approach to compose generic view models (each capturing a particular functionality) to a given base model. He points out the problems encountered when using relationships to express composition directives, and explains why parameterization might overcome some of these problems. *Mahoney et al.* (5) focus on the specification of aspects using state charts. Using aspect-oriented techniques, the authors show how to define abstract statecharts that can later be woven into other statecharts, thus making behavior models reusable. Their approach bases on the reinterpretation of certain events of one statechart in the other statecharts. *Reina et al.* (10) inspect numerous existing aspect-oriented modeling approaches and observe that most of them are closely related to particular aspect-oriented programming platforms. They propose to rise the level of abstraction of aspect models to platform-independent models (PIMs) in the Model-Driven Architecture (MDA) [8]. In order to express each aspect PIM most appropriately, aspect-specific profiles or meta-model extensions should be used.

List of Position Papers

(1) Barra, E., Génova, G., Llorens, J., *An Approach to Aspect Modeling with UML 2.0*
(2) Han, Y., Kniesel, G., Cremers, A.B., *A Meta Model and Modeling Notation for AspectJ*
(3) Kamalakar, B., Ghosh, S., *A Middleware Transparent Approach for Developing CORBA-based Distributed Applications*
(4) Li, J., Houmb, S.H., Kvale, A.A., *A Process to Combine AOM and AOP: A Proposal Based on a Case Study*

(5) Mahoney, M., Bader, A., Elrad, T., Aldawud, O., *Using Aspects to Abstract and Modularize Statecharts*
(6) Muller, A., *Reusing Functional Aspects: From Composition to Parameterization*
(7) Nakajima, S., Tamai, T., *Lightweight Formal Analysis of Aspect-Oriented Models*
(8) Navarro, E., Letelier, P., Ramos, I., *UML Visualization for an Aspect and Goal-Oriented Approach*
(9) Park, D., Kang, S., *Design Phase Analysis of Software Performance Using Aspect-Oriented Programming*
(10) Reina, A.M., Torres, J., Toro, M., *Separating Concerns by Means of UML-Profiles and Metamodels in PIMs*
(11) Spies, E., Rüger, J., Moreira, A., *Using i* to Identify Candidate Aspects*
(12) Steimann, F., *Why Most Domain Models are Aspect Free*
(13) Tessier, F., Badri, L., Badri, M., *Towards a Formal Detection of Semantic Conflicts Between Aspects: A Model-Based Approach*

3 Results of Discussion Groups

A primary goal of the workshop was to provide a platform for researchers to discuss the impact of aspect-oriented software development on software model engineering, and vice versa. Therefore, a major part of the workshop was spent debating on important aspect-oriented modeling issues. In order to maximize productivity, the participants split into three discussion groups, each of six to eight persons. The results of the discussion are summarized in the following subsections.

3.1 Reasons for Aspect-Oriented Modeling

During the workshop, we observed that different participants had different motivations and expectations regarding the new area of aspect-oriented modeling. One of the contributions of this workshop was to identify and discuss the benefits of expressing aspects at software modeling level. Many participants agreed that aspect-oriented modeling is important because it expresses crosscutting structures and behavior at a higher level of abstraction than aspect-oriented code. However, several other interesting opinions were expressed. In particular, people expect aspect-oriented modeling to provide means for:

- *Resolving conflicts in software models*. The idea is to use aspects at modeling level to allow designers detect and resolve conflicts at early stages of the development process. Typically, code-level conflicts that result from weaving processes or aspect compositions should be detected and resolved at early stages, not at execution time.
- *Modeling reusable business rules*. The idea is to express business rules as aspects in software models that can be reused for various systems, and at different levels of abstraction.
- *Model evolution and maintenance.* Similar to programming level aspects, modeling aspects need to provide mechanisms for modularizing crosscutting concerns in software models in order to facilitate both the evolution and maintenance of models.

- *Expressing reusable functions.* The idea is that aspects can be used to express reusable functions, such as annotation diagrams for performance measurements.
- *Managing requirements.* The idea is that requirements captured from different stakeholders are naturally entangled; they should be expressed as separate aspects of the system at hand. People hope that advancing aspect-oriented modeling can help separate, combine and/or manage requirements.

To achieve these expectations much research needs to be done. We hope that submissions to future workshops will address some of the above issues.

3.2 Aspect-Oriented Modeling and Terminology

At the workshop, one discussion dealt with the terminology in aspect-oriented modeling and its relation to the terminology in aspect-oriented programming. Some of the participants thought that aspect-oriented software development in general and aspect-oriented modeling in particular could benefit from the definition of an aspect-oriented vocabulary. Terms such as "aspect", "join point" and "weaving" might have similar, yet slightly different meanings at different levels of software development – similar to what happens in object-orientation: High-level analysis objects are not identical to programming language-level objects.

Firstly, the definition of "aspect" from the programming language-level was looked at: an aspect at the programming language-level is a modularized implementation of a concern that otherwise (in a non-aspect-oriented implementation) would crosscut the program structure (or its behavior). The two key elements in this sentence are "modularize" and "crosscut". Most participants agreed that an aspect at the model-level is a modularized model of a concern that otherwise (in a non-aspect-oriented model) would crosscut the main model structure. Some participants, however, interpret the word "aspect" more like what others call "concern", and hence believe that especially at the early stages of software development there are no such things as "crosscutting aspects". At that level, every concern is a first-class citizen, so to speak. It was discovered that the particular conception of "aspects" often differs considerably depending on the abstraction level that researchers are working on (cf. section 3.3, as well).

Then the discussion moved on to try and define the term "join point", which was initially suggested to designate at modeling-level all those points at which models can be merged / composed / woven together. Unfortunately, the term "join point" was deemed problematic, as it is coined by AspectJ, coming with a well-defined meaning that may cause considerable misconception when used in the modeling domain. "Join points" in the modeling domain commonly refer to a more generalized concept, such as "some points where aspects can hook onto" (for example). The question arose whether or not "join points" should be named differently in the modeling domain to highlight this distinction.

Likewise, the term "weaving" was put to question, and opposed to "composition". "Composition" of models has been known in software modeling for a long time, and the knowledge gained in this area of research helps the aspect-oriented modeling community to specify and assess the effects of model-"weaving". However, it remains unclear if both terms may be used as true synonyms, or whether "weaving" is a special kind of "composition".

Unfortunately, no general consensus was reached for any of the terms "aspect", "join point", and "weaving". While one part of the participants believed that using an aspect-oriented vocabulary throughout the software development life cycle would benefit the aspect-oriented software development community, the other participants deemed it too early to try and define these terms in a concise way.

3.3 Aspects in the Modeling Process

During the discussion it became manifest that in the conventional software development process – i.e., in requirements engineering, analysis, design, and implementation – different aspects appear in different phases and on different levels of abstraction. Requirement level aspects such as maintainability or reusability are specified during early stages of software development, for example, while other issues such as caching or synchronization seem to be rather implementation level aspects that cannot easily be traced back to particular requirements in all cases. Finally, there are concerns that are present throughout the entire software lifecycle, e.g., security, persistence, or auditing, which are most likely to take different forms during development. At one level, they might be modeled as aspects, i.e. their model crosscuts the main model structure, but on other levels they do not.

It turns out that expectations for, and problems of, aspect-oriented modeling often differ considerably between development phases and abstraction levels. Differences exist, for example, in what should be modeled as an aspect, what should be regarded as a join point, how and when aspects should be composed with the primary model, etc. Therefore, authors were asked to clearly indicate at which level of abstraction, or phase of software lifecycle, their work is situated in order to avoid misunderstandings during the discussion.

3.4 Aspect-Oriented Modeling and UML Model Elements

One major issue in the discussion groups was to elucidate to what extent existing UML model elements are capable of expressing aspect-oriented concepts, and/or why they fail to do so. The UML elements of interest were: UML classes, UML packages, UML collaborations, UML use cases, UML templates, UML (2.0) components, ports and connectors, OCL statements, as well as *sets* of UML diagrams (e.g., class diagrams, collaboration diagrams, state charts, etc.). These model elements were collected from previous work on aspect-oriented modeling and current workshop submissions, supplemented by brain storming in the morning session of the workshop, and then discussed by each individual group.

Everyone agreed that current UML model elements were deemed to lack important characteristics of aspects:

The UML class construct provides a module for encapsulating state and behavior, and therefore seems suitable to model an aspect. Abstract classes, for instance, could be used to encapsulate partial behavior, which can then be completed and reused in subclasses. However the class construct alone is not sufficient to model the encapsulated behavior; additional UML diagrams such as state diagrams or sequence diagrams are needed. Also, a class cannot expose required interfaces, or join points, to the rest of the system.

The new UML 2.0 component element offers that possibility to expose required interfaces. This feature could be used to explicitly declare a component's join points as part of its interface. This means, however, that aspects can only hook onto such declared join points, which makes it hard to handle unforeseen extensions.

UML sequence diagrams were briefly discussed regarding the new fragments feature introduced in UML 2.0, which might make it possible to declare potential extension points.

UML templates allow a modeler to expose interfaces, and to reuse partial model elements and configure them to his / her needs. Unfortunately, standard UML templates are not powerful enough. All currently known approaches that use templates to model aspects had to extend the template mechanism and step outside the UML.

Finally, UML packages were looked at. They are the most general UML model element, since they can contain any other diagram.. They provide a nice means for separating and grouping together all elements related to a certain concern, and therefore do a great job in modularizing an aspect. Their capabilities, however, are limited when trying to model the weaving, i.e. showing how elements of an aspect affect external entities.

To summarize, aspects in aspect-orientation were identified to meet much of the characteristics of the previously mentioned UML elements, such as:

- being the encapsulation of (some structural and/or behavioral) properties,
- being first-class entities that can exist on their own right,
- being instantiatable classifiers that can have multiple instances, each having its own state, etc.

However, essential differences between aspects in aspect-orientation and existing model elements in the UML were identified as well. For example, aspects were identified to:

- provide introspection capabilities (e.g. pointcuts) and intercession capabilities (e.g. pieces of advice or introductions),
- provide a mechanism to define (extrinsic) properties of other elements,
- break encapsulation of other elements, etc.

Concerning the introspection capabilities of aspects, possible solutions were sketched during the discussion: Pointcuts, such as in AspectJ for example, were identified as declarative expressions of introspection. These expressions could be looked at as *patterns* that are matched against the elements of the base system. Looking for ways to express patterns in the UML, the participants identified UML templates and the Object Constraint Language (OCL) as possible solutions. Adaptations to these approaches were deemed necessary, though, as UML templates are designed to generate – rather than match – model elements, while the OCL is not capable of expressing timely patterns (such as denoted by AspectJ's *cflow* construct), for example. Possible help on how to deal with introspection capabilities on the modeling level could also be found in the MOF specification, as it already contains reflective capabilities.

In conclusion, the discussed ideas were deemed to be still very immature, and therefore need to be further investigated. Finding appropriate modeling solutions that can address all issues mentioned above was considered subject for future research.

3.5 Aspect-Oriented Modeling and Weaving of Models

Another important discussion topic dealt with the question how to weave models, and what model elements should serve as join points. In general, weaving on the modeling level was deemed to be more powerful than weaving on the implementation level: Firstly, weaving on the modeling level can use any model element as join point (rather than being restricted to some Join Point Model as in AspectJ, for example). Secondly, weaving on the modeling-level brings aspect-orientation to just any platform, and hence makes aspect-oriented development independent of the availability of an aspect-oriented compiler for the programming language used for implementation.

A general problem of weaving pertains to the correctness of models. Since aspects break up encapsulation of the primary model elements, weaving is most likely to change their semantics. Therefore, means must be found to identify and resolve weaving conflicts, and research in validation and testing of woven models is essential.

Besides that, parallels between weaving of models in aspect-oriented modeling and the transformation of models in the Model-Driven Architecture (MDA) have been pointed out. In a MDA context, weaving was considered to be a horizontal model transformation – rather than a vertical model transformation (which was considered to represent a refinement). The Query View Transformation (QVT) language was deemed to bring much benefit to the specification of weaving.

Another matter of interest was to identify the point in the software development process when an aspect is woven with the rest of the model (e.g., at the requirements elicitation phase, during the design phase, or not until the implementation phase). Currently there seems to be no general rule that determines the ideal time for weaving. Some aspects might be woven at an early stage, some aspects only at implementation-time, depending on the kind of aspect and the specific application is applies to. The workshop participants expressed the desire to find more general criteria, or heuristics, that would provide a guidance that helps developers to determine when weaving of which aspects should be accomplished, at what point in time, and in what order.

Finally, the issue of symmetric models vs. asymmetric models was raised [4].. In an asymmetric model, there exists a base model of the system under development that captures its main functionality. At weave time, aspects get woven into the base model, and hence the weaving transformation is asymmetric. In symmetric models, no distinction is made among different models because of the concern they address. There is no base, and hence the weaving transformation is symmetric.

Not much time was spent on discussing these two models, but it seems like the differences are similar to the ones identified at the programming language level, for example when comparing AspectJ [6] to Hyper/J [11]. The asymmetric model is easier to work with on a conceptual level. It simplifies identifying aspects since they crosscut the base model. Also, the asymmetric model can be seen as an add-on to standard object-oriented modeling. The symmetric model is appealing because of its simplicity, but might require a complete rethinking of the way we do modeling.

4 Concluding Remarks and Outlook

The purpose of this paper is *not* to provide a complete and widely accepted opinion on aspect-oriented modeling of all the authors, organizers, and workshop participants. Our intention is to give an essential input for future research on aspect-oriented modeling, pointing thus researchers to current problems and possible matters of interest. To do so, the authors' goal was to draw a full picture of all topics that have been discussed at the workshop.

There are several active research groups in the aspect-oriented community and the software model engineering community working on theoretical and practical aspect-oriented modeling issues. However, as the workshop discussions have shown, there is still lots of interesting work to be done to make aspect-oriented modeling cover the whole software development lifecycle. In particular, we need to make use of a widely accepted vocabulary; to provide well-defined modeling elements for "aspects"; to define a standardized way of identifying "join points", and supporting "weaving" mechanisms, while allowing modelers to evaluate and validate alternative aspect-oriented designs. Workshops such as this one can play a major role in addressing the above modeling issues.

At the end of the workshop, the participants were asked to provide a list of important questions to be looked at in a near future. They will be considered when establishing the agenda for envisioned successor workshops. The identified questions were:

- What is the benefit of using aspect-oriented modeling? What are the reasons for using it in the context of each software development phase?
- What prior art applies to aspect-oriented modeling? What can be learned from its origins, e.g. object-oriented abstraction mechanisms, model composition and transformation, and techniques using reflection?
- How can modeling notations visualize aspect-specific peculiarities? For example, how can we depict aspect-oriented introspection and intercession capabilities?

Acknowledgements

We'd like to thank everybody who helped to make this workshop a success: In particular, the members of the organization committee who could not make it to the workshop, i.e., Faisal Akkawi, Omar Aldawud, Grady Booch, Tzilla Elrad and Jeff Gray; the members of the program committee, Mehmet Aksit, Atef Bader, Siobhán Clarke, Bill Harrison, Hoda Hosny, Karl Lieberherr, Peri Tarr and Aida Zakaria. Finally, we would like to thank all the submitters and participants.

References

[1] Baar, Th., Strohmeier, A., Moreira, A., Mellor, St., Proc. of 7th International Conference on the Unified Modeling Language 2004, Lisbon, Portugal, October 10-15, 2004

[2] Brito, I., Moreira, A., *Integrating the NFR Framework in a RE Model*, Early-Aspects Workshop at 3rd International Conference on Aspect-Oriented Software Development 2004, Lancaster, UK, March 22, 2004

[3] France, R., Kim, D.K., Georg, G., Ghosh, S., *An Aspect-Oriented Approach to Design Modeling*, in: IEE Proc. – Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, to appear in 2004

[4] Harrison, W., Ossher, H., Tarr, P., *Asymmetrically vs. Symmetrically Organized Paradigmes for Software Composition*, TR RC22685, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA, December 2002

[5] Jackson, D., Shlyakhter, I., Sridharan, M., *A Micromodularity Mechanism*, in: Proc. of 9th International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001, pp. 62-73

[6] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G., *An Overview of AspectJ*, in: Proc. of the 15th European Conference on Object-Oriented Programming 2001, Budapest, Hungary, June 18-22, 2001, pp. 327-353

[7] Navarro, E., Ramos, I., Pérez, J., *Software Requirements for Architectured Systems*, in: Proc. of 11th International Conference on Requirements Engineering 2003, Monterey, CA, September 8-12, 2003, pp. 356-366

[8] OMG, *MDA Guide*, Version 1.0, OMG Document omg/2003-05-01, May 2003

[9] OMG, *MOF 2.0 Core Final Adopted Specification*, OMG Document ptc/03-10-04

[10] OMG, *UML 2.0 Infrastructure Specification*, *UML 2.0 Superstructure Specification*, OMG Documents pct/03-09-15 and ptc/03-08-02

[11] Tarr, P., Ossher, H., Sutton, S., *Hyper/J: Multi-Dimensional Separation of Concerns for Java*, in: Proc. of the 24th International Conference on Software Engineering 2002, Orlando, Florida, May 19-25, 2002, pp. 689-690

[12] *The 5th International Workshop on Aspect-Oriented Modeling*, Homepage, List of Position Papers, and Schedule, http://www.cs.iit.edu/~oaldawud/AOM/index.htm

[13] Yu, E., *Modeling Strategic Relationships for Process Reengineering*, PhD Thesis, DKBS-TR-94-6, Department of Computer Science, University of Toronto, 1995

Software Architecture Description and UML

Paris Avgeriou[1], Nicolas Guelfi[1], and Nenad Medvidovic[2]

[1] Software Engineering Competence Center (SE2C), University of Luxembourg, 6, rue Richard Coudenhove-Kalergi, L-1359, Luxembourg
{paris.avgeriou, nicolas.guelfi}@uni.lu
[2] Computer Science Department, School of Engineering, University of Southern California, Los Angeles, CA 90089-0781 U.S.A
neno@usc.edu

Abstract. The description of software architectures has always been concerned with the definition of the appropriate languages for designing the various architectural artifacts. Over the past ten years, formal or less formal Architecture Description Languages (ADLs) and supporting methods and tools have been proposed by researchers. More recently, UML has been widely accepted in both industry and academia as a language for Architecture Description (AD), and there have been approaches to UML-based AD either by extending the language, or by mapping existing ADLs onto it. The upcoming UML 2.0 standard has also created great expectations about the potential of the language to capture software architectures, to allow for early analysis of systems under development and to support qualities. Furthermore, the latest trends such as MDA and the aspect-oriented paradigm are tightly connected with both UML and AD, thus promoting new approaches which combine the two. This workshop attempted to delve into this multi-faceted field, by presenting the latest research advances and by facilitating discussions between experts.

1 Introduction

Industry and academia have reached consensus that investing in architectural design in the early phases of the lifecycle is of paramount importance to the project's success [2, 4, 5, 7, 10]. Moreover an undoubted tendency to create an engineering discipline in the field of software architecture is apparent if we consider the published textbooks, the international conferences devoted to it, and recognition of architecting software systems as a professional practice [4]. Despite the attention drawn to this emerging discipline, there has been little guidance regarding how to describe a software architecture. Evidently there have been advances in the field, especially concerning design and evaluation methods, as well as reusable architectural artifacts such as architectural patterns and frameworks. And there is growing consensus nowadays about certain aspects of the task of software architecture description, such as the satisfaction of stakeholders' concerns through multiple views [1, 5]. But a software architecture needs to be rigorously described if we expect to benefit from its advantages such as communication of stakeholders, early analysis of the system, support of qualities and trouble-free maintenance. Unfortunately the problem of describing software architectures has not been solved; on the contrary we are still at early stages of addressing it [4].

One of the greatest challenges in describing software architectures, and a 'hot' topic of research nowadays, is the definition of the appropriate languages. The past ten years, formal or less formal Architecture Description Languages (ADLs) and supporting methods and tools have been proposed by researchers [8]. More recently, the Unified Modeling Language is being de facto accepted in both industry and academia as a language for Architecture Description (AD), and there have been approaches of UML-based AD either by extending the language, or by mapping existing ADLs onto it. The upcoming UML 2.0 standard has also created great expectations about the potential of the language to capture software architectures, to allow for early analysis of systems under development and to support qualities. Furthermore, MDA and the Aspect-Oriented paradigm are tightly connected with both UML and AD, thus promoting new approaches which combine the two.

The workshop on Software Architecture Description and UML made an effort to look into these issues from a holistic viewpoint inside the UML community. It has brought together researchers and practitioners who work on diverse aspects of Architectural Description (AD) of software systems, related to the Unified Modeling Language. It thus fostered a presentation of the latest approaches on the field from both industry and academia, as well as a creative discussion between the participants in specific themes.

The rest of this workshop report is organized as follows: Section 2 presents the theme of the keynote speech, which discussed the upcoming UML 2.0 standard with respect to specifying and enforcing software architectures. Section 3 outlines the contents of the papers that were presented in the workshop and involved the issues of components, connectors, architecture-based analysis, static and dynamic modeling of architectures with UML 2.0 and a development methodology that combines Aspect-Oriented Modeling and MDA. Section 4 describes the findings of the discussion panel that consisted of three invited experts, who discussed architectural issues from the viewpoint of three respective qualities: security, mobility and performance. Finally Section 5 concludes with a brief synopsis of the state-of-the-art and future trends.

2 Specifying and Enforcing Software Architectures

The keynote speech[1] concerned the theme of *specifying* and *enforcing* software architectures during the development and evolution of the system. Current software architecting practice often fails in both these activities: first, architecture is not explicitly specified, which results in architectural intent being 'hidden'

[1] The keynote speaker, Bran Selic, is an IBM Distinguished Engineer and the chair of the OMG task force, responsible for the finalization of the UML 2.0 standard.

or possibly 'buried' inside the code; second, as a result of this lack of architecture specification, the architecture cannot be enforced, i.e. it cannot be properly implemented and maintained. It thus runs the risk of getting corrupted by developers that don't understand it, even by minor changes such as bug fixes. This results in 'architectural decay', where the system implementation gradually drifts apart from the original architectural intent.

Architectures are meant to be modeled at different levels and different languages, including the code level in a programming language. Models can therefore be refined continuously at various levels of detail, from different viewpoints, until the system is fully specified at the code level. In this respect, a software system is distinguished from other engineering products, by the unique characteristic that the model per se *evolves* into the system implementation. The model-driven development paradigm implements this principle, based on two complementary techniques: *abstraction* that is supported by modeling languages, and *automation* of the transformations between the models, that is provided by tools. Thus, enforcing the architecture can be much more straightforward, since the architectural decisions can be passed on to the system through code generation. The benefits are increased productivity and assured quality, since it will then be impossible to corrupt the architectural intent by low-level programming.

The following definition of an *engineering model* was proposed: "a reduced representation of a system that highlights the properties of interest from a given viewpoint". This definition emphasizes the following aspects: that the model is an *abstraction* of the system at a specific level of detail; that it is often looked upon from different *viewpoints* that demonstrate different sides of the system; and that *representing* the system is not merely "syntactic sugar" but a meaningful visual aid. A software architecture in particular is a model that enables communication between the different stakeholders, drives the construction of the system and determines the system's capacity for evolution growth.

The rest of the discussion focused on the run-time view of software architectures, which deals with the run-time organization of significant software components interacting through interfaces, and being composed of successively smaller components and interfaces. The application of UML 2.0 in describing run-time architectures was elaborated. First, it was stressed that run-time architectures should not be modeled only statically through class diagrams but also at an instance level through collaborations. Then the most fundamental new concept in the upcoming UML 2.0 standard for architectural description was discussed: *structured classes.* These are originated from Architecture Description Languages and describe the inner structure of a class, either through a behavior specification or through a collaboration of parts through connectors. It is highly recommended that architects use structured classes to describe the hierarchical decomposition of systems' run-time structures. Ports are also a key concept in structured classes, since they are points of grouped interactions, they specify provided and required interfaces, and they decouple the structured class from external entities. Structured classes are joined by connectors through their ports, and connectors in turn can be constrained by a specific behavior protocol that can be appropriately specified with the use of interaction diagrams. The importance of structured classes lies in the fact that they can be rigorously specified and thus facilitate code generation in order to *enforce* the architecture. Finally, the Component element has been "upgraded" in UML 2.0 to subclass Structured Class and to allow for mappings to specific platforms (e.g. EJB).

As a concluding remark, it was stressed that "to architect is to model". The process of architecting is inherently a modeling activity which captures the architectural intent and subsequently enforces it during system development and evolution, thus preventing 'architectural decay'. Model-driven technologies are a promising approach in the software architecture field, and UML 2.0 in particular, encapsulates much of what was defined in classical architectural description languages and also supports architectural enforcement.

3 Issues in Software Architecture Description with UML

In order to facilitate the presentation of key topics in the field and to allow for extensive discussion on them, only six papers were selected to be presented to the workshop. The papers were chosen through a rigorous reviewing process, aimed at singling out high-quality submissions that concern a wide gamut of research issues: components, connectors, architectural analysis, architecture description in industrial projects, behavioral modeling and new trends such as Aspect-Orientation and MDA.

3.1 Documenting Architectural Connectors with UML 2

The paper by Ivers et al. discusses the issue of UML 2 support for Architectural Connectors, a concept which is treated by the software architecture community as a first-class entity, just like components. The authors recollect that UML 1.x was an awkward fit in representing architectural connectors, which led to designers making their own conventions, either by using the existing UML elements, or by extending the language. There was much anticipation in the architecture community to see whether the upcoming UML standard would provide a better support for connectors. The authors examine the concept of connectors in UML 2 with respect to how well it satisfies 4 criteria:

– *semantic match* - connectors naturally signify pathways of interaction.
– *visual clarity* - connectors should be distinguishable from components and be represented by a minimum number of visual elements.
– *completeness* - connectors should be able to represent behavior, state and interfaces.
– *tool support.*

The authors briefly analyze to what extent these criteria are fulfilled by four standard UML 2 elements, which could be used as connectors, namely Associations, Association Classes, Classes and Connectors. Their findings are that none of these elements is a perfect match, instead there is a tradeoff in using each one of them. The authors conclude that even though UML 2.0 is much more apt for architectural documentation in several aspects, representing connectors still seems to be problematic. It must be noted that the analysis presented in this paper focused on standard UML elements, and not on extensions of the language.

3.2 Using UML for SA-Based Modeling and Analysis

The paper by Cortellessa et al. reports on how their research group is using UML to specify Software Architectures (SA) for different kinds of analysis. They outline four different approaches related to SA-based model-checking, testing, performance and reliability analysis respectively:

– *Model checking* - It is performed through the *Charmy* framework that aims to assist the software architect in designing Software Architectures and in validating them against functional requirements. Formal model checking techniques are used to check the consistency between the SA models and functional requirements. The description of the architecture is based on stereotyped class diagrams for the component and connector view, state machines for the component behavior and scenarios for the specification of temporal properties.
– *Testing* - It aims to check to what extent a system under implementation conforms to its architectural specification. It offers the advantage of testing early and at a higher-level of abstraction. It allows the detection of structural and behavioral problems from UML stereotyped class diagrams and state diagrams respectively, as well as the specification of test cases as sequence diagrams. Test cases are firstly specified at an architectural level and then refined into the code level.
– *Performance analysis* - It is achieved through the SAPone approach which automatically generates a performance model, based on a Queueing Network model (QN), from a SA specification described by UML 2.0 Diagrams. The UML profile for Schedulability, Performance and Time (SPT) is utilized in order to annotate the UML diagrams with performance-related information.
– *Reliability analysis* - It focuses on modeling the reliability of a system as a function of the reliability of individual components and connectors. The authors have proposed an extension of UML to represent concepts in the reliability domain, especially for component-based systems, and thus produce *reliability models* at an architectural level.

Finally the authors introduce their ongoing work which aims to provide a framework for incorporating all the above approaches into the same analysis framework. Their rationale is based on the need to tradeoff between functional and non-functional properties, by integrating the analyses of individual properties. They have introduced a framework that aims at such an analysis integration, *independently* of the notations or languages used for the different kinds of properties.

3.3 Flexible Component Modeling with the ENT Meta-model

The paper by Brada identifies two problems in current component meta-models: (i) they merely reflect the present state-of-the-art in component technology without allowing for extensions that could accommodate future developments; (ii) the visual languages associated with the meta-models, similarly, offer specific, preset views on components rather than more adaptable visualizations. The author proposes an approach in order to alleviate both these deficiencies:

– by introducing the ENT component meta-model which is open to future technological developments and which enables us to define the component characteristics from the users point of view (rather than in just technological terms). This meta-model is built upon an analysis of a number of research and industrial component meta-models.
– by proposing a flexible graphical notation that, based on the meta-model abstractions, allows the users' to adjust the visual representation of component interfaces. This concept is similar to using multiple views for showing different aspects of a system's architecture.

The author advocates that this approach would allow present or future component metamodels to be mapped to the ENT metamodel, even if such mappings always entail semantic gaps. Finally, the combination of components specified in this metamodel with architectural connectors would be a challenging field of future research.

3.4 Designing the Software Architecture of an Embedded System with UML 2.0

The paper by Frick et al. discusses the results of an industrial project for model-driven development of embedded systems software. Part of this methodology was to devise an architecture description language, based on selected elements of UML 2.0, particularly leveraging the port concept. The authors focused on describing the software architecture of embedded systems as interconnections of modules through explicitly-specified *provided* and *required* interfaces. Pairs of required and provided interfaces are perceived as *contracts* that the module must conform to, and they are usually attributed to the module's ports. Thus a module imports or exports a service specified by a contract, through a port. Furthermore a module implementation can be either: (i) a behavioral model in terms of a state machine that implements the module services; (ii) a composite module that has an internal structure as mandated in the UML 2.0 composite structures package; (iii) code written in a programming language and wrapped in the context of UML. The first two cases support code generation, therefore, all three implementations are considered executable.

Another significant aspect of this approach is that it aims at product-family architecture design, where individual products, or *variants* are specific configurations of module variants. Subsequently the latter are different implementations of the same interface. This is a very helpful concept in the development of embedded systems, as environment components can be considered as variants and they can be simulated in order to test embedded control software.

3.5 Behaviors Generation from Product Lines Requirements

The paper by Ziadi et al. draws upon the current research trend to model variability in Product Lines (PL). Related research work has so far concentrated on the *static* architecture of PL; the authors extend it to the *behavioral* aspects. In specific the authors propose an approach to derive the behavioral specification of individual products from that of a Product Line. To begin with, they exploit the ability of UML 2.0 to algebraically compose sequence diagrams through special composition operators. Therefore, they specify PL behavioral requirements as algebraic expressions extended with constructs to specify variability. Building on that, they synthesize the detailed behavior for each product member in the PL in two stages: The first stage uses abstract interpretation of the variability operators in scenarios to get behavior specialization of the PL according to given decision criteria; in the second stage, the resulting product behavior specifications, expressed as sequence diagrams, are synthesized into statecharts.

This approach thus helps to refine behavioral specifications for the whole product family, which are specified in high-level sequence diagrams, into product-specific implementation-level statecharts. Therefore it fosters efficient, formalized traceability between requirements on a Product Line level and detailed design of individual products in PL. It can also promote reuse of statecharts between products that share common behavior.

3.6 A UML Aspect-Oriented Modeling Approach for Model-Driven Software Development

The paper by Vachon and Mostefaoui introduces a development methodology that combines Aspect-Oriented Modeling and Model-Driven Architecture (MDA), which have both received growing interest from the research community. The authors claim that these approaches naturally complement each other: MDA separates the business model, the computation model and platform specific-design decisions into distinct development steps and documents the transformation from one to another; aspect-orientation separates core functional requirements from “crosscutting application concerns” while at the same time merging them in a clean and explicit manner. Consequently, combining the two approaches entails the ‘weaving’ of aspects in the different MDA models and supporting the transformations among them.

Their method supports an iterative stepwise refinement process that not only takes care of the satisfaction of functional requirements in an MDA fashion, but also introduces aspects early: these are woven into platform-independent design decisions and then transformed to platform-specific models. From the aspect-orientation side, the authors propose a UML Profile as a modeling notation, called Aspect-UML, for the specification of aspects and their join points. From the MDA side they present the MDA-based development phases, focusing particularly on the transformation of platform independent models (PIM) into platform specific models (PSM). In specific they explain how to transform Aspect-UML models into selected PSM, using a mapping between their corresponding metamodels. They also explain how new generation transformation tools can potentially automate the transformation of an Aspect-UML PIM into target PSM.

4 Architectural Support for Qualities

The aim of the discussion panel was to discuss critical, but under-addressed issues pertaining to software architectural description. Three distinguished experts were invited to the discussion in order to shed some light on the architectural support for 3 respective qualities, namely security, mobility and performance. The short talks of the experts and the subsequent discussions are summarized in the following paragraphs.

Dr. Jan Jürjens[2] explored the field of architectural design for security-critical systems [6]. Dr. Jürjens advocated that the main problem in the software architecture of security-critical system is that security is not designed up-front as an architecture-level issue, but rather "circumvented" at a later stage, resulting in potential security compromises. The remedy that is proposed for this problem, is an approach, entitled *Model-based Security Engineering*. It deals with architectural design artifacts arising in industrial development of security-critical systems (e.g. UML models) and requires tool-supported security analysis. It mandates the automatic analysis of models against security requirements and then follows a round-trip engineering style, where code or tests are generated from models and vice versa. The approach suggests the use of UML for the typical reasons of standardization, broad industry adoption and extensive tool support. A UML profile, named UMLsec, has been proposed in order to grasp the details of secure systems development. Finally this approach suggests the use of *secure architectural patterns* in a formal, methodological way, using the aforementioned UMLsec profile.

Professor Raffaela Mirandola[3] elaborated on the issue of mobility of software systems. She advocated that mobility of code is an architectural-level design issue that serves several goals such as service customization, dynamic functionality extension, fault-tolerance, performance improvement etc. Unfortunately there is no silver bullet in designing architectures of mobile systems, instead there is always the risk of performance shortcomings. She also explained that the current architectural styles for mobile systems can be classified into two categories: those where only code moves and those where code moves along with its state. As far as the *locations* where mobility of software takes place, they can be logical or physical, they can be nested, and finally they can also be mobile themselves. There are currently two approaches to modeling architectures of mobile systems

[2] Dr. Jürjens is affiliated to the Technical University of Munich, Germany. email: juerjens@in.tum.de

[3] Professor Mirandola is affiliated to Universita di Roma "Tor Vergata", Italy. email: mirandola@info.uniroma2.it

[9]: (i) UML-based modeling which is visual, extensible, a de-facto standard in industry for architectural design but has imprecise semantics; (ii) "mobility-oriented" Process Algebras which are unambiguous, have compositional features, and facilitate analysis, but are overly complex, not widely used and they lack support for architectural design. Typically, as in other cases, bridging between formal and semi-formal approaches is a key research issue in this area.

Professor Murray Woodside[4] tackled the issue of performance modeling [11] with respect to software architecture modeling. He stressed the fact that the relation between the performance model and the architecture is bi-directional: the performance model is constructed upon the architectural model, and the results of performance modeling are a valuable feedback in selecting and validating the various design choices in the architectural model. Performance modeling is actually based on architectural high-level information such as architectural styles, partitioning of functionality into components etc., but it also requires additional low-level details, such as workload and demands for operations. Analysis of performance models subsequently takes place through formal techniques such as queueing, petri nets, layered queueing, simulation etc. An interesting aspect that arises from performance modeling is that different architectural configurations can be compared against each other, as long as some parameters such as workload, the platform and the number of processors are kept invariant. However, it is of paramount importance to evaluate the tradeoff between the detail and accuracy (and therefore cost) of performance modeling and the value of the produced results. A useful rule of thumb in this case is to match the precision of performance data to the level of detail in the architecture model.

5 Epilogue

We can safely conclude that the description of software architectures is still a very relevant subject in the research community. The practice of software architecting is growing, and there are many notations used in the scope of architectural description. UML is gaining more and more prominence and has made steps forward in this direction but can still be awkward to use for certain aspects of architectural description. The support for qualities has been under-represented in ADLs in the past, and this has not changed with UML; nor will the use of UML per se provide such support. A synergy between experts in the domains of the various qualities and software architecture, is a challenging issue and a necessity. The UML 2.0 standard is currently being explored for its appropriateness in the field, while some shortcomings have already been identified and attempts are made to overcome them. Nevertheless UML 2.0 is likely to redraw the landscape substantially. We are looking forward to this development, and will gauge UML 2.0's native architectural support, and software engineering community's reactions to it in deciding on possible follow-ons to this workshop.

[4] Professor Woodside is affiliated to Carleton University, Canada.
email: Murray.Woodside@sce.carleton.ca

References

1. Avgeriou, P., Guelfi, N., Razavi, R.: Patterns for documenting software architectures. Proceedings of the 9th European Pattern Languages of Programming (EuroPLOP) conference. July 2004, Irsee, Germany.
2. Bosch, J.: Design and Use of Software Architectures. Addison-Wesley, 2000.
3. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M.: Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. John Wiley and Sons, 1996.
4. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond. Addison-Wesley, 2002.
5. IEEE, Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE std. 1471-2000, 2000.
6. Jürjens, J.: Secure Systems Development with UML. Springer-Verlag 2004.
7. Kruchten, P.: The 4+1 view model of architecture. IEEE Software, November 1995.
8. Medvidovic, N. and R. N. Taylor: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Trans. Softw. Eng. 26 (2000), pp. 70-93.
9. Grassi, V., Mirandola, R., Sabetta, A.: A UML Profile to Model Mobile systems. In Proc. of UML 2004 conference, 11-15 October 2004, Lisbon, Portugal. Springer LNCS 3273.
10. Shaw, M., Garlan, D.: Software Architecture - Perspectives on an emerging discipline. Prentice Hall, 1996.
11. Petriu, D, Woodside, M.: A Metamodel for Generating Performance Models from UML Designs In Proc. of UML 2004 conference, 11-15 October 2004, Lisbon, Portugal. Springer LNCS 3273.

Appendix: Acknowledgement

We extend our thanks to all those who have participated in the organization of this workshop, particularly to the program committee, which is comprised of:

– Arsanjani Ali, IBM Global Services, USA
– Bosch Jan, University of Groningen, the Netherlands
– Dubois Eric, CRP Henri Tudor, Luxembourg
– Egyed Alexander, Teknowledge Corporation, USA
– Ewetz Hans, Clearstream International, Luxembourg
– Garlan David, Carnegie Mellon University, USA
– Issarny Valerie, INRIA, France
– Kruchten Philippe, University of British Columbia, Canada
– Ortega-Arjona Jorge, Universidad Nacional Autonoma de Mexico, Mexico
– Pastor Oscar, Universidad Politecnica de Valencia, Spain
– Poels Geert, University of Arts and Sciences Brussel, Belgium
– Razavi Reza, University of Luxembourg, Luxembourg
– Riehle Dirk, Stanford University, USA
– Romanovsky Alexander, University of Newcastle, UK
– Rosenblum David, University College London, UK
– Sharif Niloufar, Clearstream International, Luxembourg

SVERTS – Specification and Validation of Real-Time and Embedded Systems

Susanne Graf[1], Øystein Haugen[2], Ileana Ober[1], and Bran Selic[3]

[1] VERIMAG, Grenoble, France
{Susanne.Graf, Ileana.Ober}@imag.fr
[2] University of Oslo, Oslo, Norway
Oystein.Haugen@ifi.uio.no
[3] IBM, Canada
bselic@ca.ibm.com

Abstract. This paper presents an overview on the workshop on Specification and Validation of Real-time and embedded Systems that has taken place for the second time in association with the UML 2004 conference. The main themes discussed at this year's workshop concerned modeling of real-time features with the perspective of validation as well as some particular validation issues.

1 Introduction

Embedded applications have often strong constraints with respect to time related aspects. Moreover, overall systems may be huge, and even if the embedded hard real-time components are relatively small, there is some global interdependence and the existence of a global model in a uniform framework is an important issue. The Unified Modeling Language UML can play this role, even if the real-time aspects are not really integrated today in existing tools. UML aims at providing an integrated modeling framework encompassing architecture descriptions and behavior descriptions. A first step to the integration of extra functional characteristics into the modeling framework has been achieved by the "UML profile for schedulability, Time and Performance" [OMG03] and more recently a "UML Profile for Modelling Quality of Service and Fault Tolerance Characteristics and Mechanisms (QoS)" [OMG04]. One of the objectives of UML is to support the model driven approach (MDA) which consists of transforming models towards executable implementations.

In order to be able to exchange models with the aim to applying formal validation, it is important to have a common understanding of the (dynamic) semantics of the given notations in the modeling and the validation tool. Other important issues in the domain of real-time are methodology and modeling paradigms allowing us to break down the complexity and tools which are able to verify designed systems well.

The IST project Omega [Omega, GH04] aimed precisely at the definition of a UML profile for real-time and embedded systems with a semantic foundation and with tool support for validation. Some of the criteria for defining this profile were

- Taking into account the fact that validation is just one – although important - aspect of the problem, another main objective of modeling is deriving implementations. Therefore, the chosen profile should be appropriate for the domain of applications and not just for a particular validation tool.
- Fixing a dynamic semantics and a notion of consistency between notations is important in order to guarantee consistency between the validated model and the implementation.

The profile that has been developed in Omega is a rich subset of UML with some extensions: it distinguishes a time independent subset for modeling systems consisting of reactive components, for which an operational semantics has been defined [DJP*02, ZH03] and a real-time profile compatible with SPT, but which, contrary to SPT, fixes a concrete syntax and provides a semantic foundation [GOO03]. Notice that this real-time framework defines a set of constructive time constraints, expressive enough to define a precise semantics for all the time constraints introduced in SPT as tag values or stereotypes by means of constraints between well defined occurrences of *events*. Events represent time points, and we have defined naming conventions for events associated with the execution of any syntactic construct[1].

Several verification approaches and tools have been adapted to handle this profile and connected to UML tools via the XMI standard exchange format. Some of the requirements for the tools and methods were:

- To be flexible with respect to the semantic choices so as to be open for easy integration of semantic variations, at least concerning the resolution of non determinism induced by the intrinsic concurrency.
- Not to impose too strict constraints on the modeling approach and the development methodology, by nevertheless providing guidelines for the usage of tools.
- The mapping to the input language of the tool should not provide an obstacle for the use of the tool compared with modeling directly in the tools notation, meaning that a careful reflection concerning the concepts to be preserved is needed.

An overview on the Omega validation approach can be found in [Omega, GH04], where the tool, taking into account the most complete version of the profile, in particular the real-time aspects, was presented last year at SVERTS [OGO03]. The work done in this project raised a lot of questions. Concerning the handling of semantic issues in the context of UML, one question was to what extent the semantics should be fixed so that the diagrams are still able to represent an intuition that can be shared amongst different users. Another issue was to define a profile with a semantics able to take into account the different modeling paradigms used in the context of real-time and embedded systems; there we had to recognize that it is hard to model

[1] This is a similarity to UML 2.0 where a start and a finish event is associated with every behavior execution, but we have introduced a concrete syntax for these events, and we have defined a set of concrete attributes these events may have.

synchronous interaction directly[2], and this was discussed at last year's workshop. Concerning the interaction with CASE tools, the conclusion must be drawn that exchange of models between tools is not there today, and this is due to both weaknesses of the exchange format itself and of the existing tools.

The aim of this workshop was to bring together researchers from academia and industry to discuss and progress in these issues, as well as other issues in the context of time, scheduling and architecture in UML and UML related notations, such as notations for expressing time and architecture related requirements, semantic issues, analysis tool and modeling paradigms.

2 The Contributions

Seven contributions of very high quality were presented, selected from 19 regular submissions. All presentations were backed by a full paper of between 8 and 20 pages. All of the papers together with a report on the workshop's result are also published separately as a technical report at Verimag [GHOS04]. The corresponding presentation slides have been made available from the workshop website at www-verimag.imag.fr/EVENTS/2004/SVERTS. In this section, we only give summaries of each paper. The papers presented looked at the workshop's themes from very different angles.

2.1 Comparing UML Profiles for Non-functional Requirement Annotations: The SPT and QoS Profiles [BP04]

This contribution compares two of the before mentioned UML profiles adopted by OMG for annotating non-functional requirements of software systems, SPT, formally adopted in 2003 and the QoS profile. The SPT profile was the first attempt to extend UML with basic timing and concurrency concepts, and to express requirements and properties needed for conducting schedulability and performance analysis. While the SPT profile is focused on these two types of analysis, the more recent QoS Profile has a broader scope, aiming to allow the user to define a wider variety of QoS requirements and properties.

The SPT and QoS profiles are - together with the simple time model already included in UML 2.0 - the most important standardization efforts for modelling time, and a comparison is therefore important. The authors applied the two profiles to the same, rather elaborate, example – an embedded automation system.

While the QoS profile is almost UML 2.0 compliant, the SPT profile is a standard profile for UML 1.x and the UML 2.0 version has yet to be made. The authors claimed that SPT is easier to apply but is less flexible.

According to the results of the study there are mechanisms that are lacking in both profiles, and the authors have suggested improvements.

[2] It is possible to define workarounds allowing the description of an equivalent behaviour as by using a synchronous approach, but not in a direct way at the same level of abstraction.

2.2 A Formal Framework for UML Modelling with Timed Constraints: Application to Railway Control Systems [MCM04]

In the context of railway signalling systems, time related features play a relevant role at the validation process and specialists are confronted more and more with the necessity of applying formal methods as a means for preventing software faults. UML offers a standard notation for high quality systems modelling; however its lack of a standardized formal semantics explains the existence of few tools supporting analysis and verification. The authors of this contribution propose a formal support of UML model-based verification by mapping a subset of UML to time-extended ***B*** specifications [Abr96]. The main goal is to enable consistency checking through UML diagrams using existing tools for *B*. The approach is illustrated by means of the application to a railroad level crossing system, with convincing results.

UML's lack of formal semantics is a recurring theme and the common approach to remedy it is to give a transformation mapping from a subset of UML to some formal language with an existing tool support. This paper also does this. The subset of UML considered here consists of a subset of UML 1.4 state machines plus *OCL* [OMG03b] for the definition of pre- and post conditions. The formal language to which this subset is transformed is *B*. As verification using the *B* approach is an interactive process, the approach brings in some extra efforts for the designer.

2.3 On Real-Time Requirements in Specification-Level UML Models [PM04]

The design of software systems usually advances from abstract to more concrete. Unfortunately, proper specification of real-time related issues has often been postponed to the implementation phase, potentially leading to increased complexity in design. This has at least partly been due to the lack of suitable abstractions and notations for expressing real-time requirements at an abstract level, using e.g. use cases. In this paper, an approach is introduced, where use-case level behavioural specifications can be augmented with real-time properties. It is also shown that these properties can be treated as a separate issue from the underlying behaviour for e.g. eased reasoning. The verification and validation of such specifications from the viewpoint of automated tool support is briefly discussed.

Contrary to the previous paper [MCM04], the authors provide also a notation for their UML-like concept. Some have compared "*joint actions*" with formalized use cases. This may be a valid comparison, but it is also possible to see these joint actions as a new concept based on pre- and post-conditions on the same general abstraction level as use cases. TLA theorem proving [Lam94] has been applied for formal verification of the example railroad crossing model, and a mapping to timed automata and corresponding model checking by the Kronos tool for model-checking of timed automata [Yov97].

2.4 Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite [BGHS04]

Model checking of complex time extended UML (UML/RT) models is limited today due to two main obstacles: (1) The state explosion problem restricts the size of the UML/RT models which can be addressed and (2) standard model checking approaches cannot be smoothly integrated into the usually incremental and iterative design process. The presented solution for incremental design and verification with UML/RT within the FUJABA[3] Real-Time Tool Suite [BG*04] overcomes these two obstacles by applying a compositional reasoning approach that is based on a restricted notion of UML patterns and components. A mapping of a – somewhat restricted - subset of the UML/RT component model and additional real time extensions for UML state diagrams to hierarchical timed automata of *Uppaal* [LPY97] is presented which enables compositional model-checking of partial models such as patterns and components. The developed tool support makes an incremental and iterative design and verification process possible where only the patterns and components which have been modified have to be rechecked rather than the whole UML/RT model.

This approach is based on the assume/guarantee paradigm for safety properties [Pnu85] which requires decomposing global specifications into properties of patterns and components and their environments. The case study used to illustrate the approach and where it can be applied successfully, is a shuttle railroad where several shuttles may join to build temporary convoys. This approach is interesting because of its obvious practical potentials.

2.5 An Analysis Tool for UML Models with SPT Annotations [HMPY04]

This paper describes a plug-in for the Rhapsody tool, which demonstrates how simple UML models with SPT annotations can be analysed using the Times tool - a tool for modelling, schedulability analysis, and code generation for timed systems. The plug-in takes as input an UML model corresponding to a model that can be handled by the Times tool, consisting of a set of components whose behaviours are specified by statecharts with operation calls, where operations are defined by SPT timing parameters for their execution time, deadline and priority. The output is a network of timed automata extended with tasks that can be analysed using the Times tool [AF*03]. In particular, the Times tool will show whether the operations invoked from the UML model are guaranteed to meet their deadlines or not under the given assumption.

A case study is presented where the method is applied to an SPT annotated UML model of an adaptive cruise controller. The tool Times is run as a plug-in to the commercial Rhapsody UML tool.

2.6 Worst-Case Execution Time Analysis from UML-Based RT/E Applications [MGLT04]

Moving from code-centric to model-centric development seems to be a promising way to cope with the increasing complexity of real-time embedded systems. Validation is then one of the key-points of their development. Relating to this goal, schedulability analysis methods are generally used to validate a part of the system's real-time requirements. These methods rely on estimations of the Worst-Case Execution Time (WCET) of every task of the system. This paper presents some approaches to derive these WCET estimates from a detailed UML model of the application.

[3] "From UML to Java And Back Again".

The approach aims to combine a static and a dynamic approach where the static analysis finds all possible execution paths and then the dynamic analysis means selecting some of these executions and calculating WCET based on information on the execution time of the instruction set of the processor on which the system will be executed. The work is carried out in the Accord/UML modelling tool [LGT98] and the validation tool *Agatha* based on symbolic execution [Lug04]. The advantage of this approach over the usual one, consisting of measuring WCET of tasks, is that it provides over-approximations. It gives good scalability and will make it more attractive for practitioners. The disadvantage of the approach is that in its present form, without relatively precise information on the underlying platform, such as out-of-order executions and caches, the over-approximations tend to be huge. Also, it remains to be shown if in the context of object orientation architecture dependent features can be exploited in any way.

2.7 Validating UML Models of Embedded Systems by Coupling Tools [HMP04]

To support multi-disciplinary development of embedded systems, a coupling has been performed between a UML-based CASE tool (*Rose RealTime*) – used to model the embedded software - and a tool for modelling of the continuous dynamics of physical parts of the system (*Simulink*). The aim is simultaneous simulation of the software model and its environment model in both tools, thus allowing an early exploration of the possible design choices over multiple disciplines. A first prototype of the coupling has been implemented, where it turned out that achieving a common notion of time and a proper treatment of timers and data exchange was the most difficult part. To this end, a separate component is inserted as “glue” between the tools to take care of smoothing out the differences.

The work has been inspired by the need to model *Océ* copying machines where the software resides on a very intricate mechatronic system.

3 Workshop Results

Most presentations address modelling and validation or analysis of safety critical systems and more particularly real-time issues in the context of UML. The main subjects addressed in the papers and the discussions concerned the following themes.

3.1 Modelling and Semantics for Validation

Several papers address the modelling of real-time systems using an extended subset of UML for which validation support can be provided. The choice of the presented approaches was to identify a subset that could be directly mapped into the input language of some tool, providing the semantics and the validation support for this approach.

Most of the resulting frameworks propose useful modelling concepts, and at least one in use today seems to be integrated in a real development process [BGHS04]. Nevertheless they all represent partial frameworks for modelling certain aspects of systems and none of them provides a complete framework for a model based approach, where a rich model is maintained and appropriate verification models are just like the code obtained in an algorithmic way. Nevertheless, the presented profiles represent interesting aspects and may be adapted in the context of such a framework.

A clear consensus is that in the context of safety critical real-time systems, the existence of a formal semantics of all the defined concepts is needed in order to allow reasoning on the modelled systems. Nevertheless, it seems to be unclear if in the context of UML a standard semantic framework could be achieved; there are many actors and many different possible semantic choices, even in the context of real-time and embedded systems. A reasonable requirement could be that tool providers have to provide a readable description of the semantics chosen in their tool. How to obtain such "readable" semantics is an interesting research topic.

3.2 Validation and Analysis

The properties that are important in the context of real-time systems concern both functional and reactivity properties defining constraints on the duration between occurrences of events.

Functional properties may be completely time independent, but it might be useful to consider a timed model (which is often quite abstract) in order to guarantee progress properties or for systems where time is used for guaranteeing correct synchronization (e.g. through the use of timeouts).

In systems where computations are distributed or where communication times are more important than execution times, reactivity properties can often be verified on a model in which only assumptions on *durations* are made and resource constraints are abstracted.

Furthermore, schedulability of a system under a given constraint on the set of resources is verified generally on models where actions are abstracted to a duration constraint (e.g. a deadline) and an execution time constraint. Important parameters of this analysis are the execution time constraints used, and obtaining good approximations – mainly worst case execution times (WCET) – is an important topic. Results obtained in other contexts (see e.g. [TSH*03]) tend to indicate that good approximations of WCET can only be obtained in conjunction with a relatively detailed model of the platform, and, on the other hand, the dynamic aspects brought in by object orientation do not allow one to really profit from these aspects.

Finally, whenever the system under analysis comprises parts controlling a physical system with continuous behaviour, one has to analyse the correct interplay between a continuous and a digital behaviour where important properties are stability and controllability for example.

Some of the validation problems may be somehow associated with particular design phases or view points. Fixing the parameters of one analysis influences the options for the others, but there is not necessarily a predefined order in which things need to be done. Also, in the context of a model based development approach, any update of the model must allow one to redo all the validations which might be altered by the change.

The papers presented at the workshop, put forward methods for one of the before mentioned validation or analysis problems. Most of them provide semantics in terms of timed automata [AD94] or some extension of them as they provide a convenient model for combining time constraints, control flow and concurrency. The work on computation of WCET uses a simpler model by considering execution times of basic instructions as costs of transitions which have to be added so as to be able to compute the maximal cost of a set of finite executions. The work on the interaction between a continuous and a discrete model involves co-simulation between two tools providing both discretized timed executions. It does not necessitate a hybrid model encompassing both kinds of computations as it builds upon existing tools for such models.

The feasibility of validation and analysis for realistic models is an important issue. In the context of real-time systems however, the use of abstraction and compositional verification is made more difficult due to the fact that time constraints are hard to decompose. Approaches based on property decomposition can be applied only in absence of resource dependent time constraints.

3.3 UML and Safety Critical Systems

Notice that the problems induced by inheritance or dynamic evolution of the system configuration are not addressed by any of the contributions, but are mostly excluded from the considered settings. The appropriateness of object orientation for this kind of systems has been questioned a lot. Should we consider these approaches as an additional argument for this doubt?

Reuse is sometimes mentioned as one of the main arguments for object orientation, but it is rarely brought into practise. But even without reuse, object orientation has a lot of advantages concerning the structuring of a system. It is also useful, when in every instance of the system all the parameters are fixed and the configuration has a more or less) static nature, as this is required when a system has to be certified.

Clearly, an interesting research topic is to study how more dynamics can be introduced in the specifications of safety critical real-time systems without compromising static (off-line) verification.

4 Conclusions

With respect to the expression of time constraints there are two opposed trends:

1. There are those frameworks based on a small set of relatively low level but expressive concepts as they are handled in validation tools,
2. And those providing the user mainly with a set of relatively rigid patterns for the expression of time constraints. The contribution [BP04] show that even closely related profiles define redundant concepts which are even incompatible at the syntactic level.

Some effort clearly remains to be made concerning this issue.

Concerning validation of timing constraints, an important issue is to provide methodologies allowing the application of compositional methods also in a non distributed setting.

Concerning the computation of bounds of execution times of tasks, it remains to be understood in how good approximations can be obtained in an object oriented setting.

References

[Abr96] J.-R. Abrial. The B-Book: Assigning Programs to Meanings. Cambridge University Press, 1996.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. Theoretical Computer Science, 126(2):183-235, 1994.

[AF*03] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: a tool for schedulability analysis and code generation of real-time systems. Proc. of 1st International Workshop on Formal Modelling and Analysis of Timed Systems, LNCS, 2003.

[BGHS04] Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling: Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite, SVERTS 2004, in [GHOS04], 2004

[BG*04] Burmester, S., Giese, H., Niere, J., Tichy, M., Wadsack, J., Wagner, R., Wendehals, L., Zündorf, A. Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. Int. Journal on Software Tools for Technology Transfer STTT, 2004 (accepted).

[BP04] Simona Bernardi and Dorina Petriu. Comparing UML Profiles for Non-functional. Requirement Annotations: the SPT and QoS Profiles, SVERTS 2004, in [GHOS04], 2004

[DJP*02] W. Damm, B. Josko, A. Pnueli, A. Votintseva, Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML. Proc. of FMCO'02, November 5---8, 2002, Leiden, the Netherlands, LNCS Tutorials 2852.

[GHOS04] Susanne Graf, Oystein Haugen, Ileana Ober and Bran Selic, Proceedings of the Workshop on Specification and Validation of Real-time Embedded Systems, SVERTS 2004, Lisbon. Verimag research report 2004-10-x, 2004.

[GOO03] Susanne Graf, Ileana Ober, Iulian Ober. Timed Annotations with UML. In: Workshop on Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS2003), San Francisco, October 2003. accepted at STTT

[GH04] Susanne Graf, Jozef Hooman. The Omega project: Correct Development of Embedded Systems. In Proc. of European Workshop on Software Architectures, EWSA, associated with ICSE 2004, LNCS, 2004

[HMPY04] John Håkansson, Leonid Mokrushin, Paul Pettersson, and Wang Yi. An Analysis Tool for UML Models with SPT Annotations, SVERTS 2004, in [GHOS04], 2004

[HMP04] Jozef Hooman, Nataliya Mulyar, Ladislau Posta. Validating UML models of Embedded Systems by Coupling Tools, SVERTS 2004, in [GHOS04], 2004

[Lam94] L. Lamport. The temporal logic of actions. ACM Transactions on Programming Languages and Systems 16 (1994) pp. 872–923

[LGT98] A. Lanusse, S. Gérard, and F. Terrier. Real-Time Modelling with UML: The ACCORD Approach. In UML98, Beyond the Notation. Mulhouse, France. 1998.

[LPY97] K. Larsen, P. Pettersson, WangYi. UPPAAL in a Nutshell. Springer Int. Journal of Software Tools for Technology 1, 1997

[Lug97] D. Lugato, et al., Validation and automatic test generation on UML models: the AGATHA approach. Special issue of the Int. Journal on Software Tools for Technology Transfer, STTT 2004 (accepted).

[MCM04] Rafael Marcano, Samuel Colin and Georges Mariano. A Formal Framework for UML Modelling with Timed Constraints: Application to Railway Control Systems, SVERTS 2004, in [GHOS04], 2004

[MGLT04] Chokri Mraidha, Sébastien Gérard, François Terrier, David Lugato. Worst-Case Execution Time Analysis from UML-based RT/E Applications, SVERTS 2004, in [GHOS04], 2004

[OGO04] Iulian Ober, Susanne Graf, Ileana Ober. Validation of UML models via a mapping to communicating extended timed automata. 11th Int. SPIN Workshop. Barcelona, Spain, LNCS 2989, 04/2004, accepted for publication in STTT.

[OMG03] OMG. UML Profile for Schedulability, Performance, and Time, Version 1.0, formal/03-09-01, 09/2003.

[OMG03b] OMG. Object Constraint Language, version 2.0. final adopted specification, document ptc/2003-10-14, 10/2003.

[OMG04] OMG. UML Profile for Modelling Quality of Service and Fault Tolerance Characteristics and Mechanisms. Specification, ptc/2004-06-01, 06/2004.

[Omega] The homepage of the Omega project can be found at http://www-omega.imag.fr/

[PM04] Risto Pitkänen and Tommi Mikkonen. On Real-Time Requirements in Specification-Level UML Models, SVERTS 2004, in [GHOS04], 2004

[Pnu85] A. Pnueli. In Transition from Global to Modular Temporal Reasoning about Programs, in Logics and Models for Concurrent Systems, NATO, ASI Series F, Vol. 13, Springer Verlag, 1985

[TSH*03] St. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, Ch. Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real Time Avionics. Proc. of the Int. Performance and Dependability Symposium (IPDS), 2003.

[Yov97] Sergio Yovine. Kronos: A verification tool for real-time systems. In the Int. Journal on Software Tools for Technology Transfer, STTT 1 (1997) 123–133

[ZH03] M. van der Zwaag, J. Hooman. A Semantics of Communicating Reactive Objects with Timing. In Proc. of Workshop on Specification and Validation of UML models for Real-Time Embedded Systems, SVERTS 2003, technical report Verimag 2003/10/22, accepted at STTT

Essentials of the 3rd UML Workshop in Software Model Engineering (WiSME'2004)

Martin Gogolla[1], Paul Sammut[2], and Jon Whittle[3]

[1] University of Bremen, Germany
[2] Xactium, Great Britain
[3] QSS/NASA Ames, USA

Abstract. This paper reports on a workshop held at the 7th UML conference. It describes motivation and aims, organisational issues, abstracts of the accepted papers, and questions raised during discussion.

1 Motivation and Aims

Model Driven Architecture (MDA) is an OMG initiative that attempts to separate business functionality specification from the implementation of that functionality on specific middleware technological platforms (e.g., CORBA, C#/DotNet, Java/EJB, XML/SOAP). This approach is intended to play a key role in the fields of information system and software engineering. MDA is supposed to provide a basic technical framework for information integration and tools interoperation based on the separation of platform specific models (PSMs) from platform independent models (PIMs). Models of low granularity and high abstraction will represent the various functional and non-functional aspects of computer systems. In the long term there will be well defined operations, implemented by commercial tools that will allow us to build, transform, merge or verify these different models. Key standards in the MDA will be based on OMG recommendations like UML, MOF, XMI, CWM, QVT.

In fact, MDA can be considered an implementation of a more general trend that has been gathering momentum in recent years called Model Driven Development (MDD). This aims to make models the primary driving assets in all aspects of software development, including system design, platform and language definition and mappings as in MDA, but also design data integration, design analysis, tool specification and product family development.

The stage is set but the effort to move from the present situation to the idyllic automatic generation of executable models for various platforms and other applications of MDD remains huge. We need to mobilize the creative energies of a very broad category of contributors, from tool builders to theoretical specialists in fields like language compilers, graph rewriting, model checking, metamodelling, and ontology engineering. We need to bring together young researchers planning to invest in this emerging new area as well as more experienced professionals with experience in areas related to automatic code generation, transformational and generative approaches or model checking.

2 Organisational Issues

This workshop is the third in a series of workshops started at UML'2002 [1] and continued at UML'2003 [2]. The paper selection process was carefully supported by an international programme committee and additional referees. All accepted papers can be found in [3].

Programme Committee

- Jean Bezivin, University of Nantes, France
- Krysztof Czarnecki, University of Waterloo, Canada
- Phillipe Desfray, Softteam, USA
- Johannes Ernst, NetMesh, USA
- Jacky Estublier, University of Grenoble, France
- Andy Evans, Xactium, Great Britain
- Martin Gogolla, University of Bremen, Germany
- Stuart Kent, Microsoft Research, Great Britain
- Steve Mellor, ProjTech, USA
- Paul Sammut, Xactium, Great Britain
- Laurie Tratt, Kings College London, Great Britain
- Jos Warmer, De Nederlandsche Bank, Netherlands
- Jon Whittle, QSS/NASA Ames, USA
- James Willans, Xactium, Great Britain

The workshop was structured into 4 thematic sessions:

- Principles, Theories and Methodologies (Papers 3.1-3.3)
- Tools and Industrial Experience (Papers 3.4-3.6)
- Model Testing and Compliance (Papers 3.7-3.9)
- Applying MDA to Ontologies (Papers 3.10-3.11)

3 Presented Papers

3.1 Towards a Basic Theory to Model Driven Engineering

Author: Jean-Marie Favre

Abstract: What is a model? What is a metamodel? What is a language? What is a transformation? How are these concepts related? It is striking to see that, though MDE is supposed to be about precise modelling, MDE core concepts are usually described in natural language or at best, using sketchy a-la UML diagrams. These diagrams are very often inconsistent and too vague to reason about. Most of the time, they are neither validated, not even used by their authors. When precise descriptions are provided, it is only to describe a specific technology. But since Model Driven Engineering is a about supporting multiple Technological Spaces (TS), the concepts of model, metamodel, and transformation should be declined, not only in the MDA TS, but also in the Grammarware TS, Documentware TS, Dataware TS, etc. To cope with these problems, we decided to start research on defining a megamodel of MDE. This paper shows how the set theory and language theory could help in understanding MDE concepts and their relationships. The megamodel could also be seen as a first version of a rudimentary theory for reasoning about MDE concepts.

3.2 A Tool Architecture Supporting Change Propagation

Authors: Marcus Alanen, Ivan Porres
This article discusses a tool architecture that supports the construction of a Model-Driven Development (MDD) tool as a collection of loosely coupled components. Each component performs a highly specialized task under the coordination of an active model repository. The key issue in this architecture is to decide how to notify changes produced by a component to the other components and when.

3.3 Are Models the DNA of Software Construction?

Authors: Friedrich Steimann, Thomas Kühne
Abstract: MDA is advocated as the next step in software construction. It builds on the availability of a powerful modelling language and model compilers that translate models into executable code. In the following, Dr. Con-known as a harsh critic of UML and as a sceptic of the feasibility of MDA- and Dr. Pro-a believer in the MDA vision-are discussing whether MDA is fundamentally flawed from the beginning or represents the most promising new development paradigm we work on today.

3.4 A Case Study on a Transformation Focused Industrial MDA Realization

Authors: Miroslaw Staron, Ludwik Kuzniarz, Ludwik Wallin
Abstract: Model Driven Architecture (MDA) is a proposal to the realization of a vision of model driven software development defined by the Object Management Group (OMG). Its adoption, nevertheless, varies in different endeavors at different companies. The presented industrial case study performed at Volvo Information Technology (Volvo IT) contributes with the evaluation of how the team studied values profiles, constraints, transformations and other elements used in their realization of MDA. In addition it provides an insight into the practical MDA based framework development process followed at that company. The findings of the study may provide a tangible basis to the forthcoming endeavors of a similar kind together with the experiences (the process) of the MDA realization which are presented in this paper.

3.5 The TopModL Initiative

Authors: Pierre-Alain Muller, Cedric Dumoulin, Frederic Fondement, Michel Hassenforder
Abstract: We believe that there is a very strong need for an environment to support research and experiments on model-driven engineering. Therefore we

have started the TopModL project, an open-source initiative, with the goal of building a development community to provide an executable environment for quick and easy experimentation, a set of source files and a compilation tool chain, and a web portal to share artefacts developed by the community. The aim of TopModL is to help the model-engineering research community by providing the quickest path between a research idea and a running prototype. In addition, we also want to identify all the possible contributions, understand how to make it easy to integrate existing components while maintaining architectural integrity. At the time of writing we have almost completed the bootstrap phase (known as Blackhole), which means that we can model TopModL and generate TopModL with TopModL. Beyond this first phase, it is now of paramount importance to gather the best possible description of the requirements of the community involved in model-driven engineering to further develop TopModL, and also to make sure that we are able to reuse or federate existing efforts or goodwill. This paper is more intended to set up a basis for a constructive discussion than to offer definitive answers and closed solutions.

3.6 Traceability Across Refinement Steps in UML Modeling

Authors: Claudia Pons, Ralf-Detlef Kutsche
Abstract: Documenting the refinement relationship between layers allows developers to verify whether the code meets its specification or not, to trace the impact of changes in the business goals and execute test assertions written in terms of abstract model's vocabulary by translating them to the concrete model's vocabulary. Refinement has been studied in many formal notations such as Z and B and in different contexts, but there is still a lack of formal definitions of refinement in semi-formal languages, such as the UML. The contribution of this article is to clarify the abstraction/refinement relationship between UML models, providing basis for tools supporting the refinement driven modeling process. We formally describe a number of refinement patterns and present PAMPERO, a tool integrated in the Eclipse environment, based on the formal definition of model refinement.

3.7 Specification-Driven Development of an Executable Metamodel in Eiffel

Authors: Richard Paige, Phillip J. Brooke, Jonathan S. Ostroff
Abstract: Metamodels precisely define the constructs and underlying well-formedness rules for modelling languages. They are vital for tool vendors, who aim to provide support so that concrete models can be checked formally and automatically against a metamodel for conformance. This paper describes how an executable metamodel - which supports fully automated conformance checking - was developed using a model-driven extension of test-driven development. The advantages and disadvantages of this approach to building metamodels are discussed.

3.8 Challenges and Possible Solutions to Model Compliance in the Context of Model Driven Development

Authors: Juan Pablo Zamora Zapata, Francis Bordeleau, Jean-Pierre Corriveau, Toby McClean
Abstract: Specifications define requirements. Numerous authors address the validation of implementations against such requirements. But there generally exists a large semantic gap between implementations and specifications, thus significantly limiting such validation approaches. Model driven development promotes the creation of a series of models to go from specification to implementation. Consequently, compliance can now be tackled between such models. In this paper, we specifically explore the challenges faced in the task of verifying the compliance of the structure of a design model against a specification model.

3.9 Running and Debugging UML Models

Authors: Miguel Pinto Luz, Alberto Rodrigues da Silva
Abstract: Software development evolution is a history of permanent searches to raise the abstraction level to new limits, thus overcoming new frontiers. Executable UML (xUML) comes this way as the expectation to achieve the next level in abstraction, offering the capability of deploying a xUML model in a variety of software environments and platforms without any changes. This paper comes as a first expedition inside xUML, exploring the main aspects of its specification including the action languages support and the fundamental MDA compliance. We also explore the model debugging capabilities as a premature means of conceptual fail discovery. In this paper is presented a new xUML tool called XIS-xModels that gives Microsoft Visio new capabilities of running and debugging xUML models. Keywords: UML, executable UML, Model Debugging, Action Language.

3.10 Approaching OWL and MDA Through Technological Spaces

Authors: Dragan Gasevic, Dragan Djuric, Vladan Devedzic, Violeta Damjanovic
Abstract: Web Ontology Language (OWL) and Model-Driven Architectures (MDA) are two technologies being developed in parallel, but by different communities. They have common points and issues and can be brought closer together. Many authors to date have stressed this problem and proposed several solutions. The result of these efforts is the recent OMG's initiative for defining an ontology development platform. However, the problem of transformation between an ontology and MDA-based languages has been solved using rather partial and ad hoc solutions, most often by XSLT. In this paper we analyze OWL and MDA-compliant languages as separate technological spaces. In order to achieve a synergy between these technological spaces we define ontology languages in terms of MDA standards, recognize relations between OWL and MDA-based ontology languages, and propose mapping techniques. In order to illustrate the approach, we use an MDA-defined ontology architecture that includes ontology

metamodel and ontology UML Profile. Based on this approach, we have implemented a transformation of the ontology UML Profile into OWL representation.

3.11 Augmenting Domain Specific UML Models with RDF

Authors: Jörn Guy Süß, Andreas Leicher
Abstract: Models are created and maintained in the context of their problem domain. Inclusion of domain rules and background knowledge by means of profiles can be complicated and demanding. This paper presents an alternative approach based on the Resource Description Framework, which attaches background knowledge and rules as present in ontologies at the fringes of the model, rather than to include them within the model. Logical reasoning and queries are thus possible in the context of ontologies and background imported from different sources into RDF. Further, the paper discusses representation of complete UML and MOF Models using RDF Schema.

4 Additional Accepted Papers

4.1 A Formal Model of Component Behaviour

Authors: Remi Bastide, Eric Barboni, Amelie Schyn
Abstract: This paper presents a component model inspired by the CORBA Component Model, and an associated formal notation based on Petri nets and dedicated to the modelling of concurrent and distributed components. The model is illustrated by a case study that illustrates its hierarchical features, and shows how the main features of components can be mapped to the constructs of the Petri net.

4.2 Modeling and Transforming the Behavioural Aspects of Web Services

Authors: Behzad Bordbar, Athanasios Staikopoulos
Abstract: This paper introduces the modeling, mapping and transformation of behavioural aspects of interacting Web services, within the context of MDA. There are certain systems, such as Web services, where the dynamic aspects are of high importance and need to be considered during the modeling and transformation process so as to create accurate representations in their target domains. To demonstrate the approach, a realistic example is presented involving a number of Web services participating in a business process expressed as choreography of exchanged messages.

4.3 Defining Model Driven Engineering Processes

Authors: Frederic Fondement, Raul Silaghi
Abstract: Software engineering techniques made it possible for developers to build larger, and more accurate, reliable, and maintainable software-intensive

systems. This was essentially possible by introducing techniques for raising the level of abstraction for describing both the problem and its solution, and by clearly establishing a methodology to define the problem and how to move to its solution. Model Driven Engineering (MDE) targets precisely at organizing such levels of abstraction and methodologies. It encourages developers to use models to describe both the problem and its solution at different levels of abstraction, and provides a framework for methodologists to define what model to use at a given moment (i.e., at a given level of abstraction), and how to lower the level of abstraction by defining the relationship between the participating models. Such an MDE process is supposed to be defined by means of assets and methodologists have the duty to provide such assets. However, it is not yet clear what exactly these assets are, despite the fact that techniques to express them have already been widely studied. This position paper addresses this issue by identifying some of the MDE assets that have to be provided, and shows how they should be defined in order to enable them to participate in different MDE process definitions.

4.4 Systematic Validation of Model Transformations

Author: Jochen M. Küster
Abstract: Like any piece of software, model transformations must be validated to ensure their usefulness for the intended application. Properties to be validated include syntactic correctness as well as general requirements such as termination and confluence (i.e., the existence of a unique result of the transformation for every valid input). This paper introduces the idea of systematic validation and then focuses on validation of syntactic correctness for rule-based model transformations.

4.5 Towards a Language for Querying OMG MOF-Based Repository Systems

Authors: Ilia Petrov, Stefan Jablonski
Abstract: This paper introduces a SQL-aligned declarative query language called mSQL (meta-SQL) for querying OMG-MOF based repository systems. Querying repository systems may be related to querying multi-database systems having a powerful data dictionary. The problem of schematic heterogeneity in multi-database, which is elegantly solved through higher order queries and the advantages they bring to various filed have been discussed in. Systems of this kind find extensive application in data-intensive Web applications, information, application and heterogeneous data source integration. Some of the key features of mSQL are: support for higher order queries and schema independent querying, unified handling of repository data and metadata, quantification over repository model elements. Additional areas to which mSQL may be applied are: querying schematically disparate models e.g. abstract schemata of components; information (schema) discovery; generic browsing of complex data collections and scientific repositories.

4.6 Mapping UML Class Diagrams to Object-Oriented Logic Programs for Formal Model-Driven Development

Authors: Franklin Ramalho, Jacques Robin
Abstract: MODELOG aims at automatically mapping UML class, object, statechart, activity and collaboration diagrams adorned with Object-Constraint Language expressions to non-monotonic, dynamic, object-oriented logic programs in Transaction Frame Logic (TFL). Coupled with the Flora-2 inference engine for TFL, MODELOG will fill five gaps in the current UML-based infrastructure for the Common Warehouse Meta-model, Model-Driven Architecture and Semantic Web visions: (1) automated data transformation transactions specified using the Meta-Object Facility for data warehousing and mining, (2) automated UML model transformations for refinement and refactoring, (3) formal verification of UML models, (4) complete UML model compiling into running code and (5) deductive and abductive inference in intelligent agents leveraging UML semantic web ontologies. In this paper, we present the MODELOG mapping of UML class diagrams to structural TFL clauses.

4.7 Improving SoC Design Flow by Means of MDA and UML Profiles

Authors: Elvinia Riccobene, Alberto Rosti, Patrizia Scandurra
Abstract: We tackle the problem of improving the SoC (System on a Chip) design flow in order to provide a modeling framework which allows exchange, reuse and integration of IP (Intellectual Property) models. In this paper, we present a UML profile of the SystemC language exploiting the MDA capabilities of defining modeling languages, platform independent and reducible to platform dependent languages. Furthermore, we discuss the advantages of high-level modeling SoC components in the style of UML using the SystemC design primitives, rather than designing at a lower level by means of coding.

4.8 An Object Oriented Model Transformer Framework Based on Stereotypes

Authors: Weerasak Witthawaskul, Ralph Johnson
Abstract: MDA modelers, like programmers in general, will develop and reuse libraries. Some of these libraries will hide details of the platforms, so the mapping from a PIM to a PSM will have to transform libraries as well. Some libraries provide common object services while others provide domain specific functionalities. These libraries will not just be class libraries, but also profiles containing stereotypes. Mercator is an extensible tool for transforming a PIM that uses platform independent libraries to a PSM. It allows model compiler developers to specify stereotype-based transformation both for transforming new libraries and for transforming libraries to new platforms. Using the Mercator framework and platform independent libraries, it becomes possible to extend the tool to migrate from one technology to another by creating new mappings for the new technology and reusing the same PIM.

5 Questions Raised During Discussion

- What do notions like modeling-in-the-small and modeling-in-the-large refer to? What is the relationship between modeling-in-the-small and programming-in-the-large?
- Will the field 'Software Engineering' emerge sometime in the future to 'Model Engineering'? Is 'Software Model Engineering' something in between?
- Are 'Models' the basic building blocks for software construction? What is the difference between modelling, specifying and programming?
- What kind of languages do we need for modeling: One big language or many small languages? In case we have many languages, how do the different languages cooperate?
- A question coming up on many events discussing 'models': How do we integrate structural and behavioral models?
- What about the importance of the UML? Will other modeling languages like Microsoft's Whitehorse initiative decrease the influence of UML? On the other hand, will it strengthen the importance of modeling languages in general?
- What is the relationship between domain specific languages and profiles for modeling languages?
- What is beyond the model? ... the metamodel? ... the megamodel? ... the megametamodel? ... the metamegamodel?

Acknowledgement

The organizers of this 3rd workshop gratefully acknowledge the initiative of Jean Bezivin and Robert France in establishing this series of workshops.

References

1. Jean Bezivin, Robert France: Proc. 1st UML Workshop in Software Model Engineering (WiSME'2002). www.metamodel.com/wisme-2002.
2. Jean Bezivin, Martin Gogolla: Proc. 2nd UML Workshop in Software Model Engineering (WiSME'2003). www.metamodel.com/wisme-2003.
3. Martin Gogolla, Paul Sammut, Jon Whittle: Proc. 3rd UML Workshop in Software Model Engineering (WiSME'2004). www.metamodel.com/wisme-2004.

Open Issues in Industrial Use Case Modeling

Gonzalo Génova[1], Juan Llorens[1], Pierre Metz[2],
Rubén Prieto-Díaz[3], and Hernán Astudillo[4]

[1] Carlos III University of Madrid
{ggenova, llorens}@inf.uc3m.es
[2] Cork Institute of Technology, Ireland
metz@cit.ie
[3] James Madison University, VA, USA
prietorx@cisat.jmu.edu
[4] Universidad Técnica Federico Santa María, Chile
hernan@inf.utfsm.cl
http://www.ie.inf.uc3m.es/uml2004-ws6/

Abstract. Use Cases have achieved wide use as a specification tool for observable behavior of systems. However, there is still much controversy, inconsistent use, and free-flowing interpretations of use case models, in fact, not even experts widely recognized in the community agree on the semantics of concepts. Consequently, use case models are dangerously ambiguous, and there is an unnecessary divergence of practice. The purpose of the workshop was to identify and characterize some sources of ambiguity. It gathered specialists from academia and industry involved in modeling use cases to exchange ideas and proposals, with an eye to both clear definition and practical application. Some presented topics were discussed in-depth (the UML metamodel for use cases, use case instances, use cases in MDD/MDA, use case model vs. conceptual model, and tools for use cases specification), while others were left as open issues for future research. We hope our suggestions will be useful to improve the metamodel of use cases, and stimulate further research to reach a stronger coupling between the use case model and other static, behavioral and architectural models.

1 Motivation and Goals

In UML there are two main representations for use cases: *textual specifications* and *diagrams*. From a methodological standpoint, these "two worlds" have been evolving in isolation to each other. A full semantic connection between use case specification items and UML use case diagrams as initially desired by Jacobson et al. in the OOSE method does not exist. This important topic is still open to discussion and agreements, and the original impetus of the workshop comes from this dichotomy between textual vs. graphical representations for use cases.

1.1 The Graphical World

UML has been aiming to formalize use cases through object-oriented semantics by declaring the metamodel element UseCase as a subtype of Classifier, which contains

Attributes, Operations and Methods, while not defining use case documentation properties or providing a tailorable use case template. This has given use cases an explicit OO formalization as desired by Jacobson et al. in OOSE. However, the absence of a reconciling explanation of this formalization with textual use case specifications as promoted by the literature, and of guidance on how to actually document use cases, has caused a certain lack of understanding among both practitioners and researchers.

The graphical world of diagrams is dominated by use case relationships. UML's explanations of *Include* and, in particular, of *Extend* remain vague, and may even seem to be contradictory. In spite of being treated asymmetrically as two different types of use case relationships, UML's explanations do not reveal any convincing distinction; precise and unambiguous definitions of terms and semantics are missing. Moreover, UML appears to mix the instance and the type view when defining the use case relationships *Include* and *Extend*.

Another aspect where UML fails to be fully clear is regarding the meaning of use case *Generalization*: there is no indication whether subtyping and/or object-oriented inheritance semantics are meant.

1.2 The Textual World

The literature commonly emphasizes and promotes written use case specifications for functional requirements capture, which are organized according to a template; there is an implicit commitment to what a use case template should include. In contrast, use case diagrams have been used merely as an adequate graphical view on, or "entry point" to, these written specifications.

Practitioners and experts in the community frequently warn against over-emphasizing use case diagrams and strenuously advise never to neglect the use case textual specifications: in practice a use case diagram serves as a support for text but not vice versa ("a bubble does not tell us the story"). Furthermore, the techniques in the textual world are much more expressive and powerful compared to the use case relationship capabilities in UML. Finally, UML does not provide graphical modeling means for many aspects used in the textual world such as linking use cases through pre- and post-condition relations.

The current literature avoids making any commitment and prefers to highlight UML's current elusive use case relationship semantics, add to these semantics, or even arbitrarily modify these semantics, thereby keeping the practical use case concepts fuzzy. Some authors even fully discourage the use of particular use case relationships, or recommend getting rid of variety and having only a single but powerful use case relationship.

1.3 Open Areas for Research

Some of the open areas identified before the workshop are:

- Alignment of textual specification and graphical representation: use case relationships, use case standard templates, use case contracts, any information missing or extra in the two representations.

- Little semantic connection between use case specification items and UML use case diagrams. In particular, UML lacks support for the connection proposed by Jacobson et al. in OOSE.
- Collaboration vs. participation among actors of a use case. Actors may have a collaborative or participatory role in a use case, yet UML diagrams do not allow distinguishing them.
- Functional vs. structural view of use cases. Use cases may be expanded to represent functional characteristics of parts of systems, yet this expansion is not possible in UML's graphical view.
- Relationships among use cases, composition: UML allows include and extend, yet composition has a different semantics; some meaningful relationships could be borrowed from other notations, such as "precedes" from OPEN/OML and "mitigates" from Misuse Cases; dependency information encoded in pre- and post-conditions cannot be depicted graphically either.

1.4 Organization

The workshop was organized by Gonzalo Génova (Carlos III University of Madrid, Spain), Juan Llorens (Carlos III University of Madrid, Spain), Pierre Metz (Cork Institute of Technology, Ireland), Rubén Prieto-Díaz (James Madison University, VA, USA) and Hernán Astudillo (Universidad Técnica Federico Santa María, Chile).

Submitted papers were reviewed by an international team of experts composed by the organizers and Shane Sendall (University of Geneva, Switzerland), Roderick Coleman (Free consultant, Germany), Wolfgang Weber (University of Applied Sciences, Darmstadt, Germany), Sadahiro Isoda (Toyohashi University of Technology, Japan), Joaquin Miller (X-Change Technologies, USA), Guy Genilloud (IT/business consultant, Switzerland), Paul Bramble (Independent consultant, USA) and Bruce Anderson (Managing Consultant, Application Innovation, IBM Business Consulting Services, UK). Each paper received between 2 and 4 reviews before being accepted.

2 Initial Positions of the Authors

The initial two sessions of the workshop were devoted to presentation of the accepted papers, which represented a good mixture of experiences and researches both from academia and industry, as was one of the goals of the workshop. The authors came to the workshop with the following positions:

- Bruce Anderson [1]. My point of view comes from practice, from wanting UML to illuminate and support my work. While use cases can usefully be considered at various levels of formality, I would like an underlying representation that allows clear semantic relationships between use cases and other artifacts, and in particular business process models, data models, test plans, system interface objects and business rules. UML should model accurately and informatively the ways in which use case models are structured, in particular for reuse and comprehensibility. This requires the metamodel to include detail at the level of steps and alternatives.

- Nelly Bencomo, Alfredo Matteo [2]. Model Driven Software tries to focus on the modeling of systems independently of the platform, then using transformations. These models should be translated to specific platforms (for example in the field of middleware/distributed applications specific platforms are CORBA, .NET, Web Services etc).
- Clay Williams, Matthew Kaplan, Tim Klinger, and Amit Paradkar [3]. We argue that use case modeling should be done in the context of a rich conceptual model. Use cases are written in terms of this model using structured natural language. We also discuss problems that arise when trying to align this representation with the UML 2.0 metamodel, including metaclass misalignment and the lack of a representation for use case content. We close by discussing four applications of our representation: prototyping, estimation, refinement to design, and test case creation.
- Michal Smialek [4]. Use cases should have precisely defined notations which are comprehensible by various groups of people in a software development project. In order to meet these diverse views, several notations are necessary. These notations should be easily transformable and should have clear mappings to other models including the conceptual model.
- Sadahiro Isoda [5]. The current UML's use-case specification has a lot of problems and even nonsense. All these problems are due to three fundamental defects originated in OOSE. These are the illusionally "actors call use cases" conjecture, mixing-up designer's simulation with real execution and poor understanding of OO. The problems can be easily solved by recognizing anew what a use case is and then modeling it guided by plain OO technology.
- Gonzalo Génova, Juan Llorens [6]. In UML, use cases are meta-modeled as classifiers. Classifiers specify a set of instances, and use case instances are said to be concrete system-actor interactions. But it is not clear how an interaction can have classifier features such as attributes, operations and associations. Therefore, we challenge the notion that use case instances are interactions. We also propose a notion of use case (a coordinated use of system operations) that is very close to the traditional protocol, therefore concluding that use cases and protocols are not essentially different things.
- Guy Genilloud, William F. Frank [7]. It is shown that that the UML ontology is unnatural (at odds with English). As a consequence, the UML standard contains numerous sentences that confuse the picture between use cases, use case instances, and use case types. It is no surprise, therefore, that many use case practitioners do not understand the Extend relationship. The ontology of the RM-ODP, on the other hand, is more natural and more easily abided by. Using it, one would explain Extend for what it is, a relationship between specifications. Following this approach is key to reconciling the diagrammatic and textual specification techniques for use cases.
- Joaquin Miller [8]. I suggest we take an indirect approach to finding techniques to specify use cases using UML: look at use cases from the ODP viewpoint; choose ODP concepts well suited to specifying a use case; find

corresponding UML constructs; adapt the UML constructs as required. I arrive at: A particular use case of a certain system is a part of the community contract of a community of a certain type. That community is represented as a UML collaboration. I discuss how that community can be specified using UML.

3 Workshop Results

The remaining two sessions of the workshop were devoted to discussions and synthesis work, trying to reach agreement wherever it was possible. We first established a list of open issues and related them to the presented papers. Then the issues related to two or more papers were discussed in-depth: the UML metamodel for use cases [3, 5, 6, 7, 8], use case instances [5, 6, 7], use cases in MDD/MDA [1, 2, 4], use case model vs. conceptual model [3, 4], and tools for use cases specification [4, 5]. Other issues related to only one paper, or not particularly related to any of the papers, were not specifically discussed, but we mention them below. The following subsections summarize the discussions and agreements about the issues discussed in-depth.

3.1 The UML Metamodel for Use Cases

This was the main issue discussed, since it was addressed more or less directly by the majority of papers, and specifically by [3, 5, 6, 7]. We agreed that the chapter devoted to use cases and the use case metamodel in the UML2 Specification [UML2] is extremely confusing, with problems that originated in Jacobson's OOSE, were handed over to UML, and then retained so long. There are several inconsistent views and interpretations of use cases, all of them supported by the UML2, and each of them having its own difficulties. Many textual explanations in the Specification do not stick to the terminology used in the metamodel itself. This is very important for tool developers that try to be compliant with the UML2 Specification: if the metamodel is inconsistent, compliance becomes an impossible task, and incompatible interpretations of use cases lead to development of tools without interoperability.

We identified specifically the following problems:

- **Use case ontology.** The UML2 Specification introduces a dichotomy between type/specification (of a thing), on the one side, and instance (the thing itself), on the other side. For use cases, the terms are "use case" for the specification and "use case instance" for the individual. However, these terms are not consistently used, since "use case" very often means "use case instance" [7, section 2.1], leading practitioners to frequent confusions.
- **Use case features.** UseCase is a specialization of Classifier, therefore it inherits Classifier's structural and behavioral features, i.e. attributes and operations. There is not a single word in the UML2 Specification that explains the meaning of use case features. If use case instances are "interactions", being each instance a sequence of message instances exchanged between the system and the actors instances as they

communicate, then we cannot understand the meaning of use case attributes and operations: what is the sense of an interaction, a collaboration among instances, having attributes and operations? [6, section 2] We cannot think of an example of this. The UML Specification should clarify this.

- **UseCase as a specialization of BehavioredClassifier.** We do not understand why UseCase specializes BehavioredClassifier instead of Behavior [3, section 3.1.1; 6, section 2], since a use case is supposed to specify a sequence of actions, that is, a behavior. We do not understand why Behavior specializes Class either.
- **System behavior, Actor behavior, or both.** There are two inconsistent views about use cases in the UML2 Specification. On the one side, it seems a use case specifies actions performed by the subject (i.e., the system to which the use case apply); on the other side, it seems a use case specifies also actions performed by the actor when communicating with the subject. It is not clear, therefore, whether the use case specifies actor behavior or not [6, section 1]. This is of great importance for deciding the contents of a use case: should actor behavior be included in the use case specification, or not? Moreover, if actor behavior is included within the use case specification, then it has no sense saying that actors are associated with use cases, and communicate with them from the outside, as the explanations in the UML Specification and the graphical notation of use case diagrams indicate.
- **Use cases as types.** The UML Specification is misleading too when it says that a use case can be used to type one of the parts or a roles in a collaboration [6, section 1]: if the use case specifies the interaction, then it cannot specify one of the parts of the interaction at the same time.
- **Use cases vs. protocol interfaces.** UML2 has introduced the concept of a protocol interface with an associated state machine with the same purpose as use cases, namely, to specify system or subsystem usages [6]. What is the difference between use cases and protocol interfaces?

We left also some open questions that can serve as starting points for future research:

- **Generalization of use cases.** What does it mean? How are use case features inherited? (This requires clarification of the notion of use case features.) Does specialization mean subtyping too? (This requires clarification of notion of use case instance.) How is use case contents inherited (pre and post conditions, action steps, state machines…)?
- **Pre and post conditions.** What is the precise meaning of pre and post conditions? When should they be checked, at run time, or at specification time? Can pre and post conditions be used to establish sequential relationships between use cases?
- **Extend and Include relationships.** Are extensions and inclusions true use cases themselves, or are they mere fragments that deserve a different name (such as system action or activity)? Should they be visible in the use case description as separate artifacts, or are they merely configured in the model in a transparent way to the user?

- **Failures and alternatives.** How do we best express failures and choices? What terminology should be used?
- **Internal behaviors/algorithms.** How do we tie business rules to action steps? What is the relationship between use cases and system operations?
- **Log-in and log-off.** Should these be considered as separate use cases on their own with some relationship to other use cases (for example, through pre and post conditions), or should they rather be considered as mere fragments within other use cases?

3.2 Use Case Instances

Another major issue that was discussed is the improper use of the term "instantiation" to refer to the performance of actions specified in a use case [5, 6, 7]. We agreed that use case instances are not "things" that execute themselves or are executed by something else, therefore use case classifiers are not specifications of sets of "things". A behavior is not a thing, it is rather something a thing does (or something a group of things jointly do). In particular, a scenario is not an instance of a use case, and a use case does not instantiate a scenario. Rather, a use case is a complex, composite action template; a use case is a description of a behavior, and a scenario is also a description of behavior: the use case considers alternatives, exceptions, etc., while the scenario is a concrete path through the use case behavior specification.

We agreed, then, that use case instances are not "things", but we did not reach a full agreement on "instantiation" being a wrong term to refer to "the performance of actions specified in a use case". If not wrong, it seems at least to be confusing. The term "use case simulation" was proposed [5, sections 3.2 and 4.3] and received with interest, but it was not accepted by everybody.

3.3 Use Cases in MDD/MDA

The integration of use cases within Model Driven Development / Model Driven Architecture requires a better definition of use case contents [1, 2, 4], specifically a better definition of use case description of behavior through sequences of action steps [3, 4], use case pre and post conditions [7], and relationship between use case model and conceptual model [1, 3, 4].

The UML2 Specification allows for several textual and graphical representations of use case behavior, but does not provide any rules for transformations between different representations at the same level of abstraction. It does not provide either any rules for transformations of these representations to other artifacts at levels closer to implementation. Obviously, a use case must allow very informal descriptions that promote a good communication with stakeholders. However, to make them more precise, we suggest use case contents can be expressed also in semi-formal ways which should not hinder communication with stakeholders and which could be used as a refinement of those informal descriptions. Specifically, we identified several different directions to achieve this goal:

- **Use case model vs. conceptual model.** Establish a clear link between the concepts employed in use case descriptions and the vocabulary employed in the conceptual model.
- **Semi-formal structure for expressing steps.** Express action steps in simple sentences with a semi-formal structure (for example, subject-verb-object). This approach has been verified in several domains with good results [4]. However, maybe this is not feasible for all domains, an issue which deserves also further research.
- **Execution semantics for action steps.** Use action steps with clear execution semantics that allow execution or simulation of the use case behavior at an abstract level (for example, input/output statements, computations, alternatives/ exceptions checking, etc.) [3].

Other closely related issues were not discussed in-depth, and we left them as open questions for future research:

- Should we extend the metamodel to achieve this level of precision, or should we rather use the profile mechanism [4]?
- Should we have one notation for use case contents, or different ones for different problem domains or different groups of people [4]?
- Should OMG focus more on defining transformations related to the Computation Independent Model and Platform Independent Model (CIM-CIM and CIM-PIM)? Should these transformations be based on more precisely defined use case model [4]?
- Pre and post conditions: how are they related to the conceptual model and to the action steps in the description of behavior?
- How do we tie the use case model to the architecture, designs, user interfaces and code that implement it? This question requires an answer imperatively, since there is currently a lot of effort and research about MDA/MDD and it is not clear how Use Cases (and its benefits) are related to the definitions of Platform Independent Models (PIMs), Platform Specific Models (PSMs) and transformations among models.

3.4 Use Case Model Versus Conceptual Model

We also provide some suggestions to tie use cases to the vocabulary they use (in the business model), since it is good practice to keep the development of the use case model and the conceptual model in step during requirements work [3, 4]:

- The conceptual model must be consistent with the terms used for information items (the vocabulary) mentioned in the use case steps, with a clear mapping between them in both directions, maybe tool-supported, and expressed with a convenient notation.
- Different stakeholders may use different terms for the same concepts, therefore a good control of synonyms is required.

A promising field for research seems to be the construction of tools for automatic transformation and synchronization of the conceptual model with the use case model. This of course is possible when use cases are described by means of a restricted language (a subset of natural language with simple syntax that can be parsed). Would such tools be worth sacrificing informal language descriptions, or a richer, more natural, syntax?

3.5 Tools for Use Case Specification

Closely related to the MDD/MDA approach also is the need to develop tools that support the development of the use case model and other related software artifacts. We propose specifically:

- The UML metamodel should be carefully re-examined so that it allows building tools that simulate use case executions. The current metamodel says too little about use case content (e.g. steps), thus making this task difficult, even impossible [5, section 4.3].
- Tools should support transformations (that have well-defined semantics) between different use case representations and between use case representations and behaviors at the design level [3, 4]. Tools should also support synchronization with the vocabulary, as was stated in the previous section.

Tool support should not limit the possibilities to represent use cases with different notations serving particular domains. Thus an observation was made that this would necessitate that the tools have certain configuration capabilities. These capabilities would allow for defining templates for use case domain-specific notations, and what is more important – templates for transformations to other models.

3.6 Other Issues

Other issues more or less loosely related to the MDD/MDA approach were not specifically discussed, and we left them as open questions for future research:

- **Use case model and requirements.** What is the difference between the use case model and the functional requirements? Do we need separate artifacts to express them? How do we tie non-functional requirements to use cases?
- **Business processes and use cases.** How are they related to each other? How can we derive system use cases from the descriptions of business processes?
- **Level of abstraction and formality.** How many levels of abstraction are desirable in a use case model? Which level of detail is it desirable to reach? What level of formality can be reasonably attained by means of natural language?
- **Context for use cases.** How do we describe the data context for a use case? How do we relate data state in inclusions or extensions to their base use case?

- **Test cases.** How do we create test cases that get the right coverage?
- **Inspection of use cases.** What kind of rules can we give to guide inspection of use cases?
- **Metrics for use cases.** What kind of metrics can we establish for use cases?

4 Conclusions and Future Work

The summary of workshop discussions shows that our main interests lie all around two main poles: a) problems posed by deficiencies in the UML2 Specification regarding use cases, i.e. *semantics* of use cases; and b) the use case model in the context of MDD/MDA, i.e. *pragmatics* of use cases. We hope our suggestions will be useful to improve the metamodel of use cases, and stimulate further research to reach a stronger coupling between the use case model and other static, behavioral and architectural models. These two issues are closely related too, i.e. the quality of the UML Specification has a practical impact: a better definition of use case semantics (metamodel) is required to achieve a wider consensus about good practices in writing use cases and to build tools that can effectively support a use case driven development (MDD/MDA); also, the UML2 metamodel should not ignore well-established industrial practices.

Publication of workshop proceedings in the *Journal of Object Technology* is scheduled for February, 2005. New versions of the accepted papers will undergo a full review process before final publication, to achieve a higher degree of unification among them.

The workshop discussions were extremely participative and fruitful, and we hope there will be similar workshops at future UML Conferences (from now on called MoDELS Conference).

More information can be found on the workshop web site.

References

1. Bruce Anderson. “Formalism, technique and rigour in use case modelling”
2. Nelly Bencomo, Alfredo Matteo. “Traceability Management through Use Cases when Developing Distributed Object Applications”
3. Clay Williams, Matthew Kaplan, Tim Klinger, and Amit Paradkar. “Toward Engineered, Useful Use Cases”
4. Michal Smialek. “Accommodating informality with necessary precision in use case scenarios”
5. Sadahiro Isoda. “On UML2.0's Abandonment of the Actors-Call-Use-Cases Conjecture”
6. Gonzalo Génova, Juan Llorens. “The Emperor's New Use Case”
7. Guy Genilloud, William F. Frank. “Use Case Concepts from an RM-ODP Perspective”
8. Joaquin Miller. “Use Case from the ODP Viewpoint”

Models for Non-functional Aspects of Component-Based Software (NfC'04)

Jean-Michel Bruel[1], Geri Georg[2], Heinrich Hussmann[3], Ileana Ober[4], Christoph Pohl[5], Jon Whittle[6], and Steffen Zschaler[5]

[1] Laboratoire d'Informatique, University of Pau,
B.P. 1155, F-64013 Pau, France
bruel@univ-pau.fr
[2] Computer Science Department, Colorado State University,
Fort Collins CO 80523, USA
georg@cs.colostate.edu
[3] Institut für Informatik, Universität München,
Amalienstraße 17, 80333 München, Germany
hussmann@informatik.uni-muenchen.de
[4] UMR Verimag, Centre Equation, 2, avenue de Vignate,
38610 Gi-Aères, France
ileana.ober@imag.fr
[5] Fakultät Informatik, Technische Universität Dresden,
01062 Dresden, Germany
christoph.pohl|steffen.zschaler@inf.tu-dresden.de
[6] NASA Ames Research Center, MS 269-2,
Moffett Field, CA 94035, USA
jonathw@email.arc.nasa.gov

Abstract. The goal of this workshop was to look at issues related to the integration of non-functional property expression, evaluation, and prediction in the context of component-based software engineering. The accepted papers looked at the issue from a very broad range of perspectives, such as development process, modelling for analysis vs for construction, or middleware componentisation. The afternoon session was completely used for discussions: discussion topics ranged from the limits of the research domain and the definition of fundamental terms and concepts to issues of compositionality.

The workshop had a very inspiring and productive atmosphere, and we are looking forward to organising future instalments of the series.

1 Introduction

Developing reliable software is a complex, daunting, and error-prone task. Therefore, many researchers are interested in improving the support for developers creating such software. Component-based software engineering has emerged as an important paradigm for handling complexity. The goal of this workshop was to look at issues related to the integration of non-functional property expression, evaluation, and prediction in the context of component-based software engineering. In this area it was our main focus to look at model-based approaches, preferably, but not limited to, UML-based approaches. This includes semantic issues, questions of modelling language definition, but also support for automation, such as analysis algorithms, MDA-based approaches, or tool-support for refinement steps.

2 Overview of Papers Presented

All of the papers together with a report on the workshop's result are also published separately as a technical report at Dresden University of Technology [2]. The corresponding presentation slides have been made available from the workshop's website www.comquad.org/nfc04. In this section, we only give summaries of each paper. All of the papers presented looked at the workshop's theme from very different angles. We will use this motivation for a first classification of the papers in the following.

2.1 Non-functional Aspects Management for Craft-Oriented Design [4]

The authors of this paper discuss a technology to combine different view-points used by different teams (or 'crafts') developing different parts of a complex application. The key idea is in the introduction of a so-called 'pivot'-element, which serves as an interface between the different models of the different teams. Each team together with project management decides which part of its models is to be public and how these models are to be represented. Thus, teams can work largely independent of each other, while still enabling project management gain a global overview of the system and ensure overall consistency. The major motivation for this work lies in the development process, namely in the support for diverse teams cooperating in a large project.

2.2 Formal Specification of Non-functional Properties of Component-Based Software [5]

This paper presents a formal specification of timeliness properties of component-based system, as an example for a formal, measurement-based approach to specifying non-functional properties. The approach is motivated by the separation of roles in component-based software development and uses separate specifications for components, containers, system services and resources. The specification is modular and allows reasoning about properties of the composed system. As the previous paper, this paper's main motivation is in the development process, however, it is more driven by the different needs of roles such as the component developer and the application assembler.

2.3 A Model-Driven Approach to Predictive Non Functional Analysis of Component-Based Systems [3]

This paper discusses an idea to use model transformation to refine a platform-independent model (PIM) into an analysis model in addition to the platform-specific model (PSM) driving development towards implementation. The authors discuss relevant relationships between the two different refinement paths (PIM to PSM and PIM to analysis model), and describe in more detail a refinement leading to a queueing network model for average-case compositional performance analysis. A major motivation for this paper is in the requirement to analyse non-functional properties of the system under development based on models of it.

2.4 Tailor-Made Containers: Modelling Non-functional Middleware Services [1]

This paper discusses the generation of tailor-made application servers from specifications of the non-functional properties to be supported. The authors suggest to model application servers as a collection of core services and aspects implementing support for individual non-functional properties. The idea is then to generate application servers from these parts, depending on the non-functional specifications of the components or applications to be executed. This paper is mostly motivated by experiences the authors made when providing runtime support for realtime properties.

2.5 OMG Deployment and Configuration of Distributed Component-Based Applications

Another important motivation for modelling non-functional properties of component-based applications lies in the actual deployment of such applications, which may necessitate reconfiguration of individual components or the complete application. As this important area had not been covered by any of the submissions, we invited Francis Bordeleau—who is one of the co-authors of the OMG deployment and configuration specification—to give a presentation on the concepts in this specification and how these relate to models of non-functional properties of component-based software. The talk was very well received. Its main message was that although the current specification does not specifically address non-functional properties, these must be considered an important ingredients in particular to reconfiguration decisions for specific target platforms or user groups.

3 Workshop Results

There are several ways that separation of concerns surfaced in the papers: The paper on craft-oriented design separated the concerns of different specialists working together towards a common goal. Achieving independence of different roles in the development process was one motivation behind the paper on formal specification of timeliness properties, while the paper on a model-driven approach to prediction separated concerns of construction vs analysis of systems. Finally, the paper on tailor-made containers separated different non-functional aspects.

A number of questions have been discussed in the afternoon sessions:

- Domain limits: This question was about defining the domain of research. In particular, we discussed terminology issues, such as what is a non-functional property?
- Where should we put support for non-functional properties? And, when in the development process do we need to consider them?
- Is there a difference between the concept of component and the concept of resource? Is there such a thing as a resource component?
- What are the different kinds of composition that are of relevance?

We'll summarize the outcome of the discussions in the following subsections.

3.1 Domain Limits

After much discussion—especially on the definition of non-functional properties—we arrived at the following understandings.

- By starting from the requirements down to the code, functional properties are those identified first.
- A property is not functional or non-functional by itself. It depends on the point of view, or intent of the system. The functionality label is dependent on the client of this property. For example, the security feature of a communication line can be functional for a certain system, but not for another one.
- Clearly there is still a wide view of what a non-functional issue is.

3.2 Support for Non-functional Properties

The consensus here was that

- Non-functional properties need to be considered throughout the entire development process.
- For verification purposes, many non-functional properties require separate analysis models to be constructed (information comes both from the development models and non-functional properties expertise) and analyzed. The results of such analyses need to feed back into the development process. Most of the time, the properties are verified at several steps of the development process and thus at several levels of abstraction. Analysis results and models can in general not be maintained over functional refinement steps, but rather must be recreated at each different level of abstraction.
- Representation of non-functional properties changes with the stage of development process. For example, properties are expressed very explicitly in the requirements, but can be represented by certain structures in the architecture, or by a middleware configuration. This requires sophisticated notions of refinement and traceability.

3.3 Resources Versus Components

Some of the talks where considering resources as components, while others separated the two concepts. This started a discussion on these notions. The general conclusion was that there is no formal difference between components and resources. However, it is practical to distinguish them for hiding implementation details and complexity. They can be distinguished based on usage, where resources represent an encapsulation of lower-level components.

3.4 Composition

We identified four different kinds of composition, but left this for further discussion:

1. What are the semantics of composition for models vs components executing in a container?
2. How do the properties of individual components contribute to the properties exhibited by the composition?

3. Is there a semantic difference between composing components for an application vs middleware components (containers)?
4. What are the semantics of composition when the constituent elements are not orthogonal? For example, how can we compose models of one component dealing with security properties of this component and with response time properties of the same component?

4 Conclusion

While non-functional properties especially of component-based software is still an open field, we did reach consensus on some points. This encourages us to continue the series of workshops.

There is still work to do:

- Industry is still waiting for an easy way to annotate models with one generally accepted and known notation for non-functional properties.
- The role of standardizing committees in this process needs to be understood. Industry and academia are expecting them to provide a framework for thinking about non-functional properties in research and application. Concrete case studies are needed to evaluate the usefulness of standards. Finally, standardization may need to be more domain-specific.

We hope to continue this series of workshops in the future.

Acknowledgements

This work has been partially funded by the French National Science Fund (FNS/ACI/ JC-9067) and the German Research Council (DFG FOR 428/COMQUAD).

References

1. Ronald Aigner, Christoph Pohl, Martin Pohlack, and Steffen Zschaler. Tailor-made containers: Modeling non-functional middleware services. In Bruel et al. [2].
2. Jean-Michel Bruel, Geri Georg, Heinrich Hussmann, Ileana Ober, Christoph Pohl, Jon Whittle, and Steffen Zschaler, editors. Proc. Workshop on Models for Non-functional Aspects of Component-Based Software. Technical report TUD-FI04-12-Sept.2004, Dresden University of Technology, 2004.
3. Vincenzo Grassi and Raffaela Mirandola. A model-driven approach to predictive non functional analysis of component-based systems. In Bruel et al. [2].
4. Francois Mekerke, Wolfgang Theurer, and Joel Champeau. Non-functional aspects management for craft-oriented design. In Bruel et al. [2].
5. Steffen Zschaler. Formal specification of non-functional properties of component-based software. In Bruel et al. [2].

OCL and Model Driven Engineering

Jean Bézivin[1], Thomas Baar[2], Tracy Gardner[3], Martin Gogolla[4], Reiner Hähnle[5], Heinrich Hussmann[6], Octavian Patrascoiu[7], Peter H. Schmitt[8], and Jos Warmer[9]

[1] University of Nantes, France
Jean.Bezivin@sciences.univ-nantes.fr
[2] EPFL Lausanne, Switzerland
Thomas.Baar@epfl.ch
[3] IBM in Hursley, United Kingdom
tgardner@uk.ibm.com
[4] University of Bremen, Germany
gogolla@Informatik.Uni-Bremen.DE
[5] Chalmers University, Gothenburg, Sweden
reiner@cs.chalmers.se
[6] University of Munich, Germany
Heinrich.Hussmann@inf.tu-dresden.de
[7] Computing Laboratory, University of Kent, United Kingdom
O.Patrascoiu@kent.ac.uk
[8] Universität Karlsruhe, Germany
pschmitt@ira.uka.de
[9] De Nederlandsche Bank, Nederland
jos.warmer@ordina.nl

Abstract. Precise modeling is essential to the success of the OMG's Model Driven Architecture initiative. At the modeling level (M1) OCL allows for the precision needed to write executable models. Can OCL be extended to become a full high-level executable language with side-effects? At the meta-level (M2), queries, views and transformations are subjects that will be vital to the success of the OMG's Model Driven Architecture initiative. Will OCL 2.0 become an essential part of the Queries/Views/Transformations standard and what will be its application areas in industry? Can the features of OCL 2.0 be used in the Model Driven Engineering (MDE) approach? This workshop aims at bringing together people from academia that are expected to report on inspiring ideas for innovative application scenarios and tools, and industrial practitioners, which are expected to provide statements on their view of the future of OCL in the context of MDE.

1 Introduction

The workshop was organized as a part of Seventh International Conference on the Unified Modeling Language <<UML>> 2004 in Lisbon, Portugal. It continued a series of OCL workshops held at previous UML conferences: York, 2000, Toronto 2001, and San Francisco 2003 and outside UML conferences in Amsterdam and Canterbury. Following the successful model of its predecessors this workshop addressed both people from academia and industrial practitioners. The aim was to provide a forum for the exchange of views on the future of OCL, to foster the identification of strategic goals for OCL and increase cooperation within OCL community.

OMG initiated in 2002 the standardization process for MOF 2.0 Query/ Views/ Transformations. In 2003, as a result of OMG's RFP, several proposals for the standardization of QVT were submitted. In this situation it is important to look ahead to the future of OCL. The main focus of this workshop was the investigation of OCL's relation with the OMG's Model Driven Architecture (MDA) framework, at the meta-model level (M2) with the future standard for QVT. There is a clear need for a high-level language to enable modelers to specify behavior at a high level of abstraction. OCL can be extended to become such an Executable UML language. An interesting question is what extensions need to be added to OCL enable this.

At the same time we solicited contributions using OCL as a constraint language on the application modeling level. Substantial progress has been achieved in this area over the last years and we encouraged in particular the submission of case studies and papers on the relation between OCL and annotation languages.

Precise modeling is essential to the success of the Model Driven Engineering (MDE) approach to develop software systems (SS). OCL can play a role at multiple levels. At the meta-level (M2), queries, views and transformations are subjects that will be vital to the success of the MDE. Will OCL 2.0 become an essential part of the Queries/Views/Transformations standard and what will be its application areas in industry?

At the modeling level (M1) OCL allows for the precision needed to write executable models. Currently OCL is restricted to side-effect free queries. Can OCL be extended to become a full high-level executable language with side effects?

How will the powerful features of OCL 2.0 be used in the Model Driven Engineering approach? Is OCL 2.0 more powerful than needed, or is not powerful enough? This workshop aimed at bringing together people from academia that are expected to report on inspiring ideas for innovative application scenarios and tools, and industrial practitioners, which are expected to provide statements on their view of the future of OCL in the context of Model Driven Engineering.

2 Objectives of the Workshop

The workshop focused on:

- Object Constraint Language and the OCL2.0 standard.
- Model Driven Engineering.
- OMG's Queries/Views/Transformations.

The objective of the workshop was to bring together a mix of leading industry, government, and university software architects, component software framework developers, researchers, standards developers, vendors, and large application customers to do the following:

- Better understand the features of OCL 2.0 and how far they go in solving problems in software industry.
- Better understand the relation between OCL and QVT.
- Identify key directions, convergence approaches and characterize open research problems and missing architectural notions in MDE.

The workshop consisted of a set of 9 invited presentations and a final discussion session. Topics of interest listed in the Call for Participation included (but were not limited to):

- OCL – the query language for Model Driven Engineering.
- Contributions to the standardization process of QVT.
- Extensions of OCL to support QVT.
- Reports on OCL or QVT case studies, tools, or applications.
- Theoretical/fundamental aspects of OCL.
- Case studies for precise modeling using OCL.
- OCL as an Executable UML language.
- Dynamic concepts in OCL.

3 Presented Papers

1. On Generalization and Overriding in UML 2.0, Fabian Büttner and Martin Gogolla, University of Bremen, Computer Science Department, Database Systems Group.

In the upcoming Unified Modeling Language specification (UML 2.0), subclassing (i.e., generalization between classes) has a much more precise meaning with respect to overriding than it had in earlier UML versions. Although it is not expressed explicitly, UML 2.0 has a covariant overriding rule for methods, attributes, and associations. In this paper, we first precisely explain how overriding is defined in UML 2.0. We relate the UML approach to the way types are formalized in programming languages and we discuss which consequences arise when implementing UML models in programming languages. Second, weaknesses of the UML 2.0 metamodel and the textual explanations are addressed and solutions, which could be incorporated with minor efforts, are proposed. Despite of these weaknesses we generally agree with the UML 2.0 way of overriding and provide supporting arguments for it.

2. OCL for the Specification of Model Transformation Contracts, Eric Cariou, Raphaël Marvie, Lionel Seinturier, and Laurence Duchien, LIFL - Université des Sciences et Technologies de Lille, UMR CNRS 8022 - INRIA Futurs, 59655 Villeneuve d'Ascq Cédex – France, {cariou,marvie,seinturi,duchien}@lifl.fr

A major challenge of the OMG Model-Driven Architecture (MDA) initiative is to be able to define and execute transformations of models. Such transformations may be defined in several ways and with various motivations. Our motivation is to specify model transformations independently of any transformation technology. To achieve this goal, we propose to define *transformation contracts*. We argue that model transformation contracts are an essential basis for the MDA, they can be used for specification, validation and test of transformations. This paper focuses on the specification of model transformation contracts. We investigate the way to define them using standard UML and OCL features. In addition to presenting the approach and some experimental results, this paper discusses the relevance and limits of standard OCL to define transformation contracts.

3. Rule-Based Simplification of OCL Constraints, Martin Giese, Reiner Hähnle, and Daniel Larsson, Chalmers University of Technology, School of Computer

Science and Engineering, 41 296 Gothenburg, Sweden, {giese, reiner, danla}@cs.chalmers.se

To help designers in writing OCL constraints, we have to construct systems that can generate some of these constraints. This might be done by instantiating templates, by combining prefabricated parts, or by more general computation. Such generated specifications will often contain redundancies that reduce their readability. In this paper, we explore the possibilities of simplifying OCL formulae through the repeated application of simple rules. We discuss the different kinds of rules that are needed, and we describe a prototypical implementation of the approach.

4. OCL as Expression Language in an Action Semantics Surface Language, Stefan Haustein and Jörg Pleumann, Computer Science Dept. VIII/X, University of Dortmund, Germany, {stefan.haustein,joerg.pleumann}@udo.edu

With the specification of Action Semantics in UML 1.5, the OMG laid ground to manipulating object diagrams in a formal way, which is a necessary prerequisite for QVT. In QVT, of course the manipulations take place at M1 level instead of M0, but due to the architecture of UML, the same mechanisms can simply be reused. Unfortunately, the Action Semantics specification does not mandate a surface language, limiting its practical application. Due to the high overlap with the Object Constraint Language, in this article we propose a surface language that is based on and aligned with OCL.

5. Disambiguating Implicit Constructions in OCL, Kristofer Johannisson, Department of Computing Science, Chalmers University of Technology and Göteborg University, S-41296 Göteborg, Sweden, krijo@cs.chalmers.se

A rule system for type checking and semantic annotation of OCL is presented. Its main feature is the semantic annotation and disambiguation of syntax trees provided by an OCL parser, in particular for implicit property calls and implicit bound variables. It is intended as a component to be plugged in to other systems that handle OCL. An implementation of the system is available.

6. Comparing Two Model Transformation Approaches, Jochen M. Küster and Shane Sendall and Michael Wahler, Computer Science Department, IBM Zurich Research Laboratory, CH-8803 Rüschlikon, Switzerland, email: {jku, sse, wah}@zurich.ibm.com

For the MDA vision to become a reality, there must be a viable means to perform model-to-model transformation. In this paper, we compare and contrast two approaches to model transformation: one is a graph transformation-based approach, and the other is a relational approach, based on the QVT-Merge submission for OMG's MOF 2.0 Query/View/Transformation Request for Proposal. We apply them both to a common example, which involves transforming UML state machines to a CSP specification, and we look at some of the concrete and conceptual differences between the approaches.

7. Composition of UML Described Refactoring Rules, Slavisa Markovic, Swiss Federal Institute of Technology, Department of Computer Science, Software Engineering Laboratory, 1015 Lausanne-EPFL, Switzerland, e-mail: Slavisa.Markovic@epfl.ch

Refactorings represent a powerful approach for improving the quality of software systems. A refactoring can be seen as a special kind of behavior preserving model transformation. The Object Constraint Language (OCL) together with the metamodel of Unified Modeling Language (UML) can be used for defining rules for refactoring UML models. This paper investigates descriptions of refactoring rules that can be checked, reused and composed. The main contribution of this paper is an algorithm to compute the description of sequentially composed transformations. This allows one to check if a sequence of transformations is successfully applicable for a given model before the transformations are executed on it. Furthermore, it facilitates the analysis of the effects of transformation chain and its usage in other compositions.

8. Embedding OCL expressions in YATL, Octavian Patrascoiu and Peter Rodgers, Computer Laboratory, University of Kent, UK , {O.Patrascoiu, P.J.Rodgers}@kent.ac.uk

Modeling is a technique used extensively in industry to define software systems, the UML being the most prominent example. With the increased use of modeling techniques has come the desire to use model transformations. While the current OMG standards such as Unified Modeling Language (UML) and Meta Object Facility (MOF) provide a well-established foundation for defining models, no such well-established foundation exists for transforming models. The current paper describes how the OCL expressions are integrated in a transformation language called YATL (Yet Another Transformation Language) to provide support for model querying. The paper presents also the transformation environment and the main features of YATL.

9. Relations in OCL, D.H.Akehurst, Computing Laboratory, University of Kent, D.H.Akehurst @kent.ac.uk

OCL is proposed as a query language within the QVT framework. The main QVT submission bases the specification of transformations on the concept of relations. Relations are not first class entities within the OCL. By extending OCL with the concept of Relations it can better serve the needs of the QVT framework. In particular this enables OCL to be used as a semantic interpretation of a QVT transformation language and may even facilitate the use of OCL as a transformation specification language.

4 Number of Participants

The workshop attracted 28 participants. There exists already a kind of "OCL community", more and more people are interested in Model Driven Engineering, and many of these people attended the UML conference series.

5 Discussion Session

The final session discussed the following topics:

- Does OCL need extensions?
- Does OCL need refactoring?

- Is it possible to embed/include OCL in other languages/systems? If yes, how hard is it?
- What is the relation between OCL and QVT?
- Has OCL been used in industry in large scale projects?

6 Organizers

- Jean Bézivin, University of Nantes, France
- Thomas Baar, EPFL Lausanne, Switzerland
- Tracy Gardner, IBM in Hursley, United Kingdom
- Martin Gogolla, University of Bremen, Germany
- Reiner Hähnle, Chalmers University, Gothenburg, Sweden
- Heinrich Hußmann, University of Munich, Germany
- Octavian Patrascoiu, University of Kent, United Kingdom (contact)
- Peter H. Schmitt, Universität Karlsruhe, Germany
- Jos Warmer, De Nederlandsche Bank, Nederland

Jean Bézivin is professor of Computer Science at the University of Nantes, France. He got his Master degree from the University of Grenoble and Ph.D. from the University of Rennes. Since 1980 he has been very active in Europe in the object-oriented community, starting the ECOOP series of conference (with Pierre Cointe), the TOOLS series of conferences (with Bertrand Meyer), the Objet'9X industry meeting (with Sylvie Caussarieu and Yvan Gallison), and more recently the <<UML>> series of conferences (with Pierre-Alain Muller). He founded in 1979, at the University of Nantes, one of the first Master programs in Software Engineering entirely devoted to Object Technology (Data Bases, Concurrency, Languages and Programming, Analysis and Design, etc.). His present research interests include object-oriented analysis and design, reverse engineering, knowledge-based software engineering, product and process modeling, model engineering and more specially the techniques of model transformation. He is a member of the ATLAS group, a new INRIA team created at the University of Nantes in relation with the LINA CNRS Lab. On the subjects of model-driven engineering and MDA(tm), he has been recently leading the OFTA industrial group in France, co-animating a CNRS specific action and a Dagstuhl seminar. He is currently involved in several EU projects.

Thomas Baar holds a diploma degree in Computer Science from Humboldt-University of Berlin and a doctoral degree from University of Karlsruhe. In his doctoral thesis, a formal semantics of OCL based on metamodeling techniques is proposed. He published about 10 papers focusing on theoretical and practical issues of OCL. Since 2003, he is a post-doc assistant at the EPFL, Lausanne, Switzerland. His current research area is specification, verification, and testing of software.

Tracy Gardner has a Mathematics and Computer Science degree from the University of Bath and a PhD in the area of programming/modelling language design which was a winner of the CPHC/BCS Distinguished Dissertations award 2000. Tracy has spent time as a practitioner of model-driven development, using the UML-based

Rational Rose Real-Time product while working for Marconi Telecommunications Ltd. Since joining IBM in 2001 Tracy has been involved in model-driven component technologies for business integration. Dr Gardner's current work is on applying Model-Driven Development to the Business Integration domain; she was the main contributor to a UML profile for automated business processes with a mapping to BPEL4WS and is now collaborating on IBM's response to the OMG's Business Process Definition Metamodel and MOF 2.0 Queries/Views/ Transformations RFPs. Tracy has presented on model-driven development at a number of industry conferences (including OMG MDA? Implementers' Workshops, Enterprise UML 2003, 1st European Conference on Model-Driven Software Engineering).

Martin Gogolla is professor for Computer Science at University of Bremen, Germany and is the head of the Research Group Database Systems. His research interests include object-oriented design, formal methods in system design, semantics of languages, and formal specification. Before joining University of Bremen he worked for the University of Dortmund and the Technical University of Braunschweig. His professional activities include: teaching computer science; publications in journals and conference proceedings; publication of two books; speaker to university and industrial colloquia; referee for journals and conferences; organizer of workshops and conferences (e.g. the UML conference); member in national and international program committees; contributor to international computer science standards (OCL 2.0 as part of UML 2.0).

Reiner Hähnle is a Professor in Computer Science at Chalmers University of Technology, Gothenburg, Sweden since 2000. He received diploma and PhD degrees in Computer Science from University of Karlsruhe in 1987 and 1992, respectively. He received a habilitation degree from Technical University of Vienna in 1997. His main research interests are non-classical logics, automated deduction, and the use of formal methods in software engineering. He authored and/or edited three books and is in the author list of over 60 publications. He wrote commissioned articles for both the Handbook of Philosophical Logic (2nd ed) and the Handbook of Automated Reasoning. He was president of the Technical Committee on Multiple-Valued Logic of IEEE CS from 2000 to 2001. He is a member of the steering committees of the IJCAR, FTP, Tableaux, and FloC conference, and co-founder of the Intl Tableaux Conference. He organized numerous workshops and conferences. In 2002, he was conference chair of CADE. Currently, he is involved in IJCAR, MDAFA, and CASSIS as PC member or invited speaker. He is member of the editorial board of Soft Computing, Multiple-Valued Logic, and QPQ (an online journal for publishing peer-reviewed source code for deductive software components). He has been involved in numerous national and international research projects as leader and grant holder. He has been reviewer for several research funding agencies such as the NSF of the US or FP6 of the EU.

Heinrich Hussmann holds a diploma degree in Computer Science from Munich University of Technology and a doctoral degree from University of Passau. He did research and education work at universities in Munich, Passau and Dresden. For several years, he was a systems engineer and team leader in the advanced development laboratory of the telecommunications division of Siemens. From 1997 to

2002 he was full professor for Computer Science at Dresden University of Technology, and since March 2003 he is full professor for Computer Science (Media Informatics) at the University of Munich (LMU). He participated in over 10 national and international projects in the area of software engineering and telecommunications, and is author of over 50 scientific publications, including three internationally published books. He is member of the program committee of the UML conferences since 1999 and a member of the steering committee since 2003. He was conference chair of the UML conference 2002 in Dresden.

Octavian Patrascoiu focused at the beginning of his academic career on programming languages and language processors. He published four books about programming languages, programming techniques and programming language processors. He also presented research papers at numerous conferences. In the last few years, he moved into the area of developing software tools for software quality assesment, software modelling and code generation. He had collaborations with software companies like IBM, Verilog, Telelogic, and TLC.

Peter H. Schmitt holds a diploma and doctoral degree in Mathematics from the University of Heidelberg. His main research contributions at that time lay in the area of Mathematical Logic and Universal Algebra. From 1985 to 1988 he worked for IBM Germany. Since 1988 he is a full professor for theoretical computer science at the University of Karlsruhe. From 1994 to 2000 he has been the chairman of the special interest group on logic and computer science of the German Computer Science Society (GI). Since 1998 he is a member of the Scientific Directorate of SCHLOSS DAGSTUHL, International conference and research centre for Computer Science. He has been involved in numerous, national and international, research projects on automated deduction and non-classical logic. He is author of some 50 scientific papers. He wrote a textbook on the theory of Logic Programming and co-edited a three-volume handbook on Automated Deduction. He is currently working in the area of formal specification and verification of programs.

Jos Warmer is one of the founders of OCL. He was responsible for OCL in the UML 1 core team and has been the leader of the OCL 2 submission team. He has been written books on UML, OCL and recently about MDA. He is a member of the programming committee of the <<UML>> series of conferences. He has been involved in organizing OCL workshops at the <<UML>> conferences, and is co-editor of the LNCS book that was the result of these workshops.

7 Conclusions

This workshop was of clear relevance to the OCL community since it discussed the future role of OCL in the MDE world. The presented papers and the final discussion lead to the following ideas:

1. OCL needs to be refactored by extending the standard library and providing a better concrete syntax.
2. The OCL2.0 standard needs to be improved to avoid misunderstandings and ambiguities.

3. OCL can be easily embedded in other languages and systems (see papers 4 and 8).
4. Both OCL and QVT share a common package of classes at the abstract syntax and semantic levels (e.g. types and expressions).
5. OCL should be used as a query language in QVT
6. OCL can be used in large-scale systems to specify constraints and contracts (see paper 2).

Critical Systems Development Using Modeling Languages (CSDUML'04): Current Developments and Future Challenges (Report on the Third International Workshop)

Jan Jürjens[1,⋆], Eduardo B. Fernandez[2],
Robert B. France[3], Bernhard Rumpe[4], and Constance Heitmeyer[5]

[1] Software & Systems Engineering, Dep. of Informatics, TU Munich, Germany
[2] Dep. of Computer Science and Engineering, Florida Atlantic University, USA
[3] Computer Science Department, Colorado State University, USA
[4] Software Systems Engineering, TU Braunschweig, Germany
[5] Naval Research Laboratory, USA

Abstract. We give a short report on the contributions to and some discussions made and conclusions drawn at the Third International Workshop on Critical Systems Development Using Modeling Languages (CSDUML'04).

1 Introduction

A *critical system* is a system in which compelling evidence is required that the system satisfies critical properties, such as real-time, safety, security, and fault-tolerance properties. The construction of high-quality critical systems, e.g., avionics systems, life-critical medical systems, weapons systems, and control systems for nuclear power plants, can be enormously difficult and costly. In recent years, many critical systems have been developed, and deployed which do not satisfy critical requirements. This has led in many cases to catastrophic system failures.

Part of the difficulty of critical systems development is that producing compelling evidence of the system's correctness can be enormously expensive. Due to high costs, producing detailed system specifications and designs along with evidence that these artifacts satisfy critical properties is normally avoided. Using UML to construct a system model that satisfies critical properties may help lower these costs since UML models are easy to understand, are amenable to mechanized analysis to check critical properties, and can be used to synthesize correct, executable code.

The workshop series on "Critical Systems Development Using Modeling Languages (CSDUML)" aims to gather practitioners and researchers to contribute

⋆ `http://www4.in.tum.de/~juerjens` . Supported within the Verisoft Project of the German Ministry for Education and Research.

to overcoming the challenges one faces when trying to exploit these opportunities. The previous editions of the series were the CSDUML'02 satellite workshop of the UML'02 conference in Dresden (Germany) and the CSDUML'03 satellite workshop of the UML'03 conference in San Francisco. Both had been very successful satellite workshops of the UML conferences. The workshop report at hand now gives an overview on the contributions for and outcomes of the CSDUML'04 workshop, which took place on October 12, 2004, as part of the UML'04 conference (October 10 – 15, 2004, in Lisbon, Portugal). It was again organized in cooperation with the pUML (precise UML) group and the working group on Formal Methods and Software Engineering for Safety and Security (FoMSESS) of the German Computer Society (GI).

In the following, we first give an overview on the various contributions to the workshop. We will then attempt to draw some conclusions on the current state of the art and future challenges in the area of the workshop.

2 Contributions

The workshop featured an invited talk with the title "On the Role of Tools in Specifying the Requirements of Critical Systems" by Constance Heitmeyer (head of the Software Engineering Section of the Naval Research Laboratory's Center for High Assurance Computer Systems and one of the internationally leading experts in the formal specification and formal analysis of software and system requirements and of high assurance software systems). Furthermore, there was a panel with the title "Providing tool-support for critical systems development with UML: Problems and Challenges" consisting of distinguished experts, which created some lively discussions on the subject.

For contributed presentations, out of a number of high quality papers submitted to the workshop, seven were selected to be presented in talks at the workshop and included as full papers in the proceedings. Three additional papers were selected to be presented as short talks and included as short papers. Furthermore, there were six posters presented at the workshop, which are included in the proceedings as abstracts. The highly selective acceptance rate again kept the workshop focused, and at a high level of quality, while allowing sufficient time for discussion.

C. Heitmeyer (Center for High Assurance Computer Systems, Naval Research Laboratory): On the Role of Tools in Specifying the Requirements of Critical Systems (Invited Talk)

In 1978, a group of researchers led by Dave Parnas developed a tabular notation for specifying software requirements called SCR (Software Cost Reduction) and used the notation to specify the requirements of a mission-critical program, the Operational Flight Program for the A-7 aircraft. Since then, the requirements of many critical programs, including control software for nuclear power plants and other flight programs, have been specified in SCR. To support formal representation and analysis of software requirements, NRL has developed a state machine model to define the SCR semantics and built a suite of tools based on this semantics for checking requirements specifications for properties of interest. Such tools are especially valuable for specifying and analyzing the requirements of software systems where compelling evidence is required that the system satisfies critical properties, such as safety and security properties. This talk described the many different roles that formally based software tools can play in debugging, verifying, and validating the requirements of critical software systems. The author's recent experience and lessons learned in specifying the requirements of a security-critical cryptographic system and two software components of NASA's International Space System were also described."

R. B. France (Colorado State University), C. Heitmeyer (CHACS, NRL), H. Hußmann (LMU Munich), F. Parisi-Presicce (George Mason University and Univ. di Roma La Sapienza), A. Pataricza (Budapest University of Technology and Economics), and B. Rumpe (TU Braunschweig): Providing Tool-Support for Critical Systems Development with UML: Problems and Challenges (Panel)

Among the panelists, and also the other workshop participants, a number of controversial issues were discussed. A selection of the discussion points and some of the opinions are given below.

Sufficient Precision of UML for Cricitical Systems Development. A range of different opinions were expressed concerning the question whether the UML is presently, or ever will be, sufficiently precisely defined to be suitable for critical systems development. The issue was raised by panelist Heinrich Hußmann who expressed doubt that this will ever be the case. Other workshop participants suggested that with UML as a family of languages, at least a core of UML might be defined in a precise way. Others expressed the concern that precision is not only necessary for critical systems development, but for any kind of advanced tool-supported use of UML, since any aspects of UML that should be supported by tools firstly have to be defined precisely enough to put one in the position to provide the tool-support.

Role of Tools for Researchers. Panelist Bernhard Rumpe raised the discussion point that researchers working on tool support for UML should consider the question what the intended use of the tools developed should be. This could start from tools as a means to validate one's ideas with respect to correctness, implementability, feasibility, scalability etc.. It could include the use of tools to demonstrate to academic peers the usefulness of one's concepts or tool-buidling as an activity helping to focus the activities of one or more research groups. A more ambitious goal would be the use as an "in-house" tool by the developers in their own projects. Finally, one could try to actually sell the tools to others who would like to use them (or give them away for free but at least sell support services). Again, a range of opinions were expressed. Bernhard Rumpe expressed concern that with academic tools intended to be used by others, this might incur a considerable effort in support and maintenance in the long run. On the other hand, panelist Connie Heitmeyer suggested that it is an important goal to try to put tools to use, for example to achieve technology transfer from research to industrial practice.

Other Issues. As part of the general discussion, several other points were raised. For example, panelist Connie Heitmeyer proposed for discussion whether UML is ideally suited specifically for describing critical requirements (as opposed to the design of critical systems), or whether a different notation might be required for that.

P. Conmy and R. Paige (University of York): Using UML OCL and MDA to Support Development of Modular Avionics Systems

A Model Driven Architecture approach to the development of Integrated Modular Avionics Systems is explained. For this, the required system behavior is captured in a Platform Independent Model using UML models that are extended with OCL constraints. The model and the constraints are then transformed to a Platform Specific Model. The paper discusses potential benefits and difficulties.

M. Huhn, M. Mutz, and B. Florentz (TU Braunschweig): A Lightweight Approach to Critical Embedded Systems Design Using UML

The paper presents a pragmatic approach to the UML based design of critical systems that has been applied in the automotive domain. For that, it focuses on so-called lightweight formal methods auch as automated static analysis and validation of dynamic behaviour by simulation (although there is a potential for also incorporating a fully formalized model analysis). The paper also presents a tool with binding to commercial CASE tools used in industry.

J. Tenzer (University of Edinburgh): Exploration Games for Safety-Critical System Design with UML 2.0

The paper presents an approach which aims to provide a smooth transition from informal UML design models to the kind of precise specifications that are needed in the formal verification of critical systems. The approach is based on exploration games played by the modeler to detect flaws and determine sources of unacceptable imprecision. As part of the game, the design is then improved. The paper discusses these ideas at the hand of UML 2.0 activity diagrams and state machines using a small critical system and gives an outlook on planned tool-support.

R. Heldal (Chalmers University of Technology), S. Schlager (University of Karlsruhe), and J. Bende (Chalmers University of Technology): Supporting Confidentiality in UML: A Profile for the Decentralized Label Model

This work has as its goal to incorporate a decentralized label model into the UML by defining a UML profile. The profile allows one to specify the confidentiality of data in UML models by annotating classes, attributes, operations, values of objects, and parameters of operations. From the annotated, code in the Java extension Jif (Java information flow) can be generated in a way which guarantees that the confidentiality constraints are not violated.

S. H. Houmb (Norwegian University of Science and Technology), G. Georg, R. B. France, and D. Matheson (Colorado State University): Using Aspects to Manage Security Risks in Risk-Driven Development

The approach presented in this paper extends the CORAS framework, which is an integrated risk management and system development process for security-critical systems based on AS/NZS 4360, RUP, and RM-ODP. In particular, it now makes use of aspects to specify security risk treatment options and to implement security mechanisms. One thus gets an aspect-oriented risk-driven development process where security requirements can be identified in each development phase. The requirements can be treated by making use of the aspects, which facilitates development and evaluation of security treatment options as well as system evolution.

M. V. Cengarle (TU Munich) and A. Knapp (LMU Munich): UML 2.0 Interactions: Semantics and Refinement

This paper is concerned with High-Level Message Sequence Charts (HMSCs) which in version 2.0 are newly integrated into UML for interaction modelling. More specifically, it considers the possibility of writing negated specifications that was introduced at the opportunity, which can be used to rule out behaviour from implementations. The paper puts forward a trace-based semantics for UML 2.0 interactions that captures the standard composition operators for UML 2.0 interactions from HMSCs, and also the added negation and assertion operators. Based on that, several alternatives for treating negation in interactions can be discussed. The proposed semantics determines whether a trace is positive, negative, or inconclusive for a given interaction.The paper also defines notions of implementation and refinement for sets of interactions based on this.

T. Massoni, R. Gheyi, and P. Borba (Federal University of Pernambuco): A UML Class Diagram Analyzer

The presented work deals with the automated analysis of UML models which include OCL constraints. More specifically, it presents an approach for the automated analysis of UML class diagrams, based on the formal object-oriented modeling language Alloy. It allows us to use Alloy's tool support for class diagrams, by applying constraint solving to automatically find valid snapshots of models. The aim of this automation is in particular to support the identification of inconsistencies or under-specified models of critical software.

I. Johnson (VT Engine Controls); C. Snook, A. Edmunds, and M. Butler (University of Southampton): Rigorous Development of Reusable, Domain-Specific Components, for Complex Applications

This paper uses a UML profile called UML-B to develop failure management systems in a model-based refinement style. It thus aims to provide rigorous validation and verification in the presence of systems in evolution. The UML-B profile can be translated into a formal notation called B-method using the U2B translation tool. It includes a constraint and action language to assist behavioral modeling. The aim is thus the reuse of reliable, domain-specific software components, in particular in avionics for safety-critical airborne systems.

Z. Dwaikat (George Mason University and Cigital) and F. Parisi-Presicce (George Mason University and Univ. di Roma La Sapienza): From Misuse Cases to Collaboration Diagrams in UML

The research presented here aims to integrate the concepts of misuse and abuse cases from software security into software engineering. The goal is to include the ability to consider abnormal scenarios into software development. More specifically, misuse behavior is described in collaboration diagrams. These are supported by a formal semantics given as positive resp. negative graphical constraints that are based on typed and attributed graphs. It allows one to detect and remove redundancies and conflicts.

B. Beckert and G. Beuster (University of Koblenz): Formal Specification of Security-Relevant Properties of User Interfaces

This paper explains how to formally model security relevant properties of user interfaces using the Object Constraint Language (OCL) at the hand of, firstly, the input-output functionality of an operating system. Secondly, using a text-based email system as an example, it is explained how input-output-related security properties of an application can be formally specified with the goal of formal verification.

E. Beato (Universidad Pontificia de Salamanca); M. Barrio-Solórzano, C. E. Cuesta, and P. de la Fuente (Universidad de Valladolid): Verification Tool for the Active Behavior of UML

This paper presents a verification approach for behavioral specifications in UML. The model-checker SMV is used to verify properties of systems specified in UML class diagrams, statecharts and activity diagrams.

M. Bujorianu (University of Cambridge) and M. C. Bujorianu (University of Kent): Towards Engineering Development of Distributed Stochastic Hybrid Systems with UML

This work investigates the possibilities to provide a bridge between systems control engineering and software engineering using UML. Specifically, it is examined how to use UML to model stochastic hybrid systems.

J. Knudsen, R. Gottler, M. Jacobsen, M. W. Jensen, J. G. Rye-Andersen, and A. P. Ravn (Aalborg University): Integrating an UML Tool in an Industrial Development Process - A Case Study

This paper reports on experiences with integrating a UML CASE tool into an industrial software development process in the field of embedded systems. As part of the process, the application documentation, test specifications and program code are kept in the tool so that the tool is fully integrated into the development process.

M. Sand (University of Erlangen-Nuremberg): Verification and Test of Critical Systems with Patterns and Scenarios in UML

This work introduces an approach for verification and test of critical hardware near embedded systems. The approach aims to provide automated tools for the simulation or verification of the models against requirements. These requirements, as well as standardized general solutions for providing them, are introduced using patterns and scenarios.

K. Tabata, K. Araki, S. Kusakabe, and Y. Omori (Kyushu University): A Case Study of Software Development Using Embedded UML and VDM

The goal of the work sketched in this paper is to design a software development method for embedded systems which combines the formal method VDM++ with Embedded UML, which is a model-based software development method that incorporates best practices and specialized processes from embedded system development.

O. Tahir, C. Sibertin-Blanc, J. Cardoso (Université Toulouse): A Semantics of UML Sequence Diagrams Based on Causality Between Actions

The goal of the work sketched in this paper is to propose a semantics for the UML sequence diagrams based on a relation of causality between the actions of emission and reception of messages. A particular interest of the research lies in the relationships between different kinds of behavioral diagrams, such as statecharts and sequence diagrams.

3 Conclusion and Future Work

As a conclusion that can be drawn from the success of the workshop, it seems that the topic of critical systems development using modeling languages such as UML is enjoying increasing attention. As was reported in several of the talks, there is an increasing number of industrial projects making increasingly sophisticated use of UML or similar notations for developing systems that have to satisfy intricate critical requirements. This shows that the various obstacles that were mentioned are not unsurmountable, but that, in fact, in various application scenarios people have managed to deal with them. Therefore, it seems that the use of UML and related notations is beginnig to reach a level of maturity which allows serious industrial use.

On the other hand, the various discussions at the workshop, during the panel session, the talks, the coffee breaks, and the workshop dinner also showed that, although the topic of critical systems development using modeling languages such as UML is seen to be a very worthwhile and timely one, many challenges still remain. The various issues that were touched and were it was felt that the final answer has not been found yet include:

– the development of sophisticated tool-support (such as automated theorem provers) for critical systems development using modeling languages such as UML,
– in particular, how to find the optimal trade-off between precision and flexibility in a notation such as UML,
– whether the use of such notations for critical systems development scales up sufficiently in an industrial context,
– and whether UML itself is suitable also for describing critical requirements (as opposed to design) or whether something else is needed.

These points already promise to again give an exciting sequel to the CSDUML workshop series in 2005. More information can be found at [CSD05].

The 2004 workshop proceedings with the contributed papers are available as [JFFR04]. As with previous CSDUML workshops, it is planned to edit a special section of the Journal of Software and Systems Modeling (Springer-Verlag) with selected contributions of the workshop. Up-to-date information on this and the workshop can be found at the workshop home-page [CSD04].

Acknowledgements. We would like to express our deepest appreciation especially to the invited speaker Connie Heitmeyer and also to the panelists and to the authors of submitted papers for their valuable contribution. We thank no less the program committee members and the additional referees for their expertise. We would also like to thank the UML'04 conference chair Ana Moreira (New University of Lisbon), the workshop chair Ambrosio Toval (University of Murcia), the local arrangements chair Isabel Sofia Brito (Politécnico de Beja), and the students involved in the organization at TU Munich (in particular Dimitri Kopjev and Britta Liebscher) for their indispensable help. In addition, some of the organizers thank their various projects (including the Verisoft project) for funding.

References

[CSD04] CSDUML 2004 webpage. http://www4.in.tum.de/˜csduml04, 2004.

[CSD05] CSDUML 2005 webpage. http://www4.in.tum.de/˜csduml05, 2005.

[JFFR04] J. Jürjens, E. B. Fernández, R. B. France, and B. Rumpe editors. Critical systems development with uml (csduml 2004). *TU Múnchen Technical Report*, 2004. UML 2004 satellite workshop proceedings.

Doctoral Symposium

Marcus Alanen[1], Jordi Cabot[2], Miguel Goulão[3], and José Sáez[4]

[1] Åbo Akademi University, Turku, Finland
maalanen@abo.fi
[2] Estudis d'Informàtica i Multimèdia, Universitat Oberta de Catalunya, Spain
jcabot@uoc.edu
[3] Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Portugal
miguel.goulao@di.fct.unl.pt
[4] Universidad de Murcia, Spain
jsaez@um.es

Abstract. The UML 2004 Doctoral Symposium was the first Doctoral Symposium in the UML Conference series. The Doctoral Symposium sought to bring together PhD Students working in areas related to UML and modeling in general. It was a full-day workshop held in parallel with the remaining workshops of the conference. Ten students were selected and were given the opportunity to present and discuss their research goals, receiving high-quality feedback from the rest of participants of the workshop, including a number of volunteer seniors that helped making the Symposium a success. The Doctoral Symposium will also be present in the next edition of the conference.

1 Introduction

The first Doctoral Symposium in the UML conference series was held on the 10th of October at UML 2004. The aim of the Symposium was to bring together PhD students working in areas related to UML and modeling in general. It was organized by five PhD Students: Marcus Alanen (Åbo Akademi University), Jordi Cabot (Universitat Oberta de Catalunya), Miguel Goulão (Universidade Nova de Lisboa), José Sáez (Universidad de Murcia) and Devon Simmonds (Colorado State University).

The Symposium was intended for students who had already settled on a specific research proposal and had some preliminary results, but still had enough time remaining before submitting their dissertation, so that they could benefit from the Symposium discussions.

Ten students were selected to participate in the Symposium. Submissions were judged on originality, significance, technical merit, presentation quality and relevance to the conference topics. Each paper was reviewed by, at least, two reviewers of the Program Committee, composed of a group of reviewers with large experience in the UML conference: Fernando Brito e Abreu, João Araújo, Jean Bézivin, Robert France, Gonzalo Génova, Martin Gogolla, Heinrich Hussmann, Ivan Porres, Bran Selic, Friedrich Steimann, Ernest Teniente, Ambrosio Toval and Belén Vela.

The selected students had the opportunity to present and to discuss their research goals, methods and results within a constructive and international atmosphere. They received high-quality feedback from the rest of participants of the workshop, including a number of volunteer seniors that attended the Symposium or part of it.

The papers presented by the selected students covered a wide range of topics, from aspects modeling to model transformation and from model validation to new methods for Information Systems development, representing most of the major topics of the main conference. All papers are available online at the Symposium web page:

`http://ctp.di.fct.unl.pt/UML2004/phdSymp.htm`

2 Structure of the Doctoral Symposium

The workshop was structured in four different sessions where we tried to group the presentations addressing related topics. Each session was composed of two or three thirty minutes presentations where each presentation included at least ten minutes for discussion.

The first session dealt with aspects in the requirements (work of Isabel Brito), analysis and design stages (work of Y. Raghu Reddy). Then we moved on to papers related with model-driven development, focused on the transformations from Platform Independent Models (PIMs) to Platform Specific Models (PSMs) and to code, either in the specific field of Pervasive Systems Development (Javier Muñoz), in the integration of Security Patterns (Diego Ray) or in the generation of databases structures to control multiplicity constraints (H.T. Al-Jumaily).

The third session included some papers on model validation (Jörn Guy Süβ, coordination diagrams (David Safranek) and model testing (Trung Dinh-Trong). Finally, two presentations proposed new methods for IS development from conceptual models. The first one proposed creating generic conceptual models for each application domain (Ruth Raventós) while the second one proposed the use of ontologies as the initial conceptual model (Jordi Conesa).

The presentations are described below, in order of presentation at the Symposium. Unfortunately two students were unable to attend the Symposium. In both cases their respective supervisor presented the work instead.

2.1 Aspect-Oriented Requirements Engineering (by Isabel Brito)

The work described focuses on a particular aspect of separation of concerns. While separating concerns into individual modules decreases software complexity and enhances understandability and traceability, there are properties which do not lend themselves to such strict modularization and are said to be cross-cutting the system. Usually they cannot be encapsulated into one component but rather are scattered throughout the system. Examples of this include non-functional requirements such as distribution, security and synchronization.

Aspect-oriented software development aims to address these issues by providing means for identification, separation, representation and composition of cross-cutting concerns. Although aspects have been used in software development, research on the use of aspects at the requirements stage is still immature. The goal of the work is to develop a framework for aspect-oriented requirements engineering that supports the identification, specification, modeling and composition of cross-cutting concerns at the requirements level.

A brief outline of how to accomplish these four tasks is given. Identifying concerns can be done using e.g. common techniques for requirements elicitation and reusing already developed catalogs. Specifying concerns is divided into four subtasks; identifying responsibilities and priorities of concerns, contributions between concerns, and dependencies among concerns. Modeling is accomplished by building a requirements analysis model as well as a behavioral model. Composition of concerns is done using match points, conflict handling and defining composition rules.

2.2 An Aspect Oriented Approach to Early Software Development (by Y. Raghu Reddy, presented by Robert France)

Locating related requirements and tracking the impact of changing requirements are two tasks that contribute significantly to the complexity in the development of large scale software systems. This work concerns the development of a rigorous aspect-oriented development approach to support these tasks during the early stages of software development.

The proposed approach is called Requirements Aspect-Oriented Modeling (RAM) and allows identifying cross-cutting concerns at the requirements level (requirements aspects). The RAM approach creates a link between requirements aspects and their corresponding design aspects by providing experience-based generic solutions that aid in the identification of conflicts between cross-cutting concerns.

The approach is expected to provide system architects with techniques to systematically identify, represent, and trace concerns throughout the software life cycle. The approach is based on the notion of viewpoints and uses the Role-Based Modeling Language (RBML) to represent cross-cutting concern solutions.

2.3 Pervasive Systems Development with the Model Driven Architecture (by Javier Muñoz)

Computing based systems growth is reaching all environments of our daily life. Pervasive systems live around us, providing services to the inhabitants of a home, the workers of an office or the drivers in a car park.

The development of this kind of systems is very hard because they have to achieve devices interoperability in a heterogeneous environment in order to satisfy system requirements. This situation requires solid engineering methods for developing robust systems. In this context, this work seeks to improve current state of the art of pervasive systems development techniques by means of an MDA based method for pervasive systems development.

It proposes the Pervasive Modeling Language (Perv-ML) a precise language for building Platform Independent Models (PIMs). Perv-ML promotes the separation of roles between analysts and architects. Analysts specify the services, structural and interaction model. Afterwards, architects specify what COTS devices and/or software systems are in charge of each service.

As a Platform Specific Model (PSM), it proposes a language for modeling an OSGi system. OSGi is a Java middleware initially created for hosting software of residential gateways. Then, it applies graph grammars to define the transformations from Perv-ML to OSGi. Finally, using a set of templates, it generates the code from the PSMs.

2.4 A Systematic Approach to Testing UML Design Models (by Trung Dinh-Trong, presented by Sudipto Ghosh)

The research proposes a systematic approach to testing design models described by UML class diagrams, sequence diagrams and activity diagrams. This approach can help in finding and removing faults in designs before these are implemented. The work suggests a dynamic testing approach in which executable forms of UML design models are exercised with test inputs. The expected behavior of a design under test is compared to the actual behavior that is observed during testing, and differences are reported.

The approach includes a set of test adequacy criteria leading to high rates of fault detection, and a technique to systematically generate test suites satisfying these criteria. It also addresses the issue of UML model execution, and the questions of how can faults be detected, and what types of faults.

None of the studied approaches for UML model execution generate test infrastructure code that supports systematic testing of models. Most of the test input generation techniques only provide test requirements without providing how to derive test inputs from the requirements. The testing approach presented can be applied to models that consist of class, sequence and activity diagrams.

The testing begins when the user provides a design under test (DUT) and a set of adequacy criteria to the system. The model is transformed into an EDUT (executable DUT) by translating it to Java, and finally, the test cases generated from the adequacy criteria and test drivers are added to the model to form the TDUT (testable DUT). Code to detect failures checks that all variables in conditions and parameters in method calls are initialized, and that the target object of a message exists. Other checks based on OCL are delegated to an external tool.

To date, the fault detection ability of the algorithm to execute and observe UML design testing has been validated using two case studies with promising results.

2.5 Integration of Security Patterns in Software Models Based on Semantic Descriptions (by Diego Ray)

This proposal aims to develop a tool-supported implementation framework for business model driven security engineering. It includes a new security engineering process that results from a fusion of software and systems engineering with security engineering and formal methods for the design and analysis of secure systems. The objective is to provide methodologies and tools for the generation of executable systems with fully configurable security infrastructures.

The proposed model is a variation of Boehm's basic spiral model for software development processes, where security engineering is integrated in the development phase of the cycle. This integration relies on the use of security patterns that represent security services with specific profiles and solutions for different environments. Each pattern will be described using XML-based meta-models. The proposal will also include the necessary pattern language.

The approach also aims to define mechanisms to automate the analysis of security-enhanced models in order to find the security patterns and to allow the inclusion of these patterns in the model, as well as defining appropriate mechanisms to validate the integrity of models with respect to security requirements.

2.6 Plugging Active Mechanisms to Control Dynamic Aspects Derived from the Multiplicity Constraint in UML (by H. T. Al-Jumaily)

This research shows how the process of transforming conceptual schemas to logical schemas in database design is sometimes subject to semantic losses. In particular, the problem is focused on the case of the cardinality constraint problem.

The issue is addressed by integrating add-ins to existing database CASE tools to automatically generate triggers which verify specific conditions on insertion, deletion and update operations in the database. The tools are used when generating the logical database schema (tables and constraints) from the UML model.

Triggers are defined according to the SQL 2003 standard, as event-condition-action rules that are activated by a database state transition. The two other important concepts are event (INSERT / DELETE / UPDATE in the database table) and activation time (BEFORE / AFTER), which defines whether the trigger is activated before or after the event.

The problem of multiplicity is split in two cases. In the first case, triggers must be generated for one-to-many associations, where two events must be considered:

1) deleting or updating in the one-table: the many-table must be cascade-deleted or updated, but no checks about multiplicity are carried out
2) deleting or updating in the many-table, minimum cardinality constraints must be checked to assure that there are enough detail items related to the master item. Insertion also needs the verification of the maximum multiplicity.

The second case deals with many-to-many associations, implemented with three tables. Here, inserting, deleting or updating in any of the three tables must be followed by the corresponding verification with the appropriate trigger.

Automatic triggers are also used to maintain the consistency in data when there are generalizations involved in the conceptual model, and these generalizations have been mapped to a three-table architecture.

The last part of the research paper describes how the add-in has been designed and integrated into the CASE tool, and explains how the user interacts with it and sets the options needed to generate the triggers.

2.7 A Standards-Based UML-Profile for Message-Based Information Dissemination (by Jörn Guy Süß)

Integrating information systems using message queues and XML documents has been considered difficult to design and manage, although they are reliable in operation. Combined with transformation technology they have been called the "preferred enterprise application integration engine". The work develops a UML profile for this domain, relying on UML and other OMG standards. Specifically, OCL is used for constraints on the UML metamodel level.

Usage of UML and modeling in general can be divided into three categories: sketches, blueprints and programs. The work looks at the viability of UML usage in blueprinting, where models are kept in a tool along with diagrams which show the different viewpoints of the system under discussion. The work first studies the available means for creating a standards-based UML profile for blueprinting business systems, and then discusses the solution in the form of the EVE framework, where OCL constraints are verified by a server separate from the actual developer's modeling environment.

Currently, the service platform, validator and local profile for analysis are implemented, while some profiles are near completion or exist only as a draft. In parallel to this development, a real industry example is modeled to ensure that the method defined by the profiles is viable.

2.8 Visual Coordination Diagrams (by David Safranek)

The objective of the work is to define a universal formal visual language for concurrent systems. Such a formalism should be capable of handling both the coordination and the behavioral aspects of concurrent systems. Properties of importance are the suitability of the formalism for modeling heterogeneity, hierarchy and component-based structure. In general, visual formalisms for concurrent systems are split into two groups. The first group is formed by state-based formalisms, e.g. statecharts, and the second one by dataflow-based formalisms, e.g. message sequence charts. Both approaches emphasize different aspects of modeled systems and can be combined in a particular design.

The proposed research strives to incorporate both the state-based and the dataflow-based approaches into a single formalism. Based on previous research, these two approaches are going to be separated into two independent layers of the language, and at both layers heterogeneity should be achieved. The proposed language, Visual Coordination Diagrams, focuses on the coordination level whereas the behavioral level is represented by state transition diagrams, formalized as a Mealy machine. Components of the system are grouped into networks at the coordination layer, where input and output ports connect components. The network structure is hierarchical.

The goal of VCD is to build a framework for the coordination of different statecharts and other visual formalisms. The development of a graphical tool for creation and modification of VCD diagrams is in progress. It will make use of current suitable verification tools.

2.9 Model and Function Driven Development (by Ruth Raventós)

This work proposes an Information System development approach that may provide a substantial increase in Information Systems development productivity and, at the same time, facilitate changes to accommodate new functional requirements. This new approach is called Model and Function Driven Development.

Its main features are the distinction between the model and the function of a Conceptual Schema (CS) and its reuse of generic conceptual schemas.

In this approach, the conceptual schema of a particular IS, called specific CS, is not generated from scratch. It is obtained by refining a generic CS of the same domain. The generic CS of a given domain consists of the elements that should be present in all or many CSs for that domain. As an example, the generic CS for an auction domain will contain all the common elements appearing in any particular CS for an auction IS, like the concepts of auction, bidder and bid.

Both, the specific and the generic CS, include two kinds of knowledge: about its domain and about the functions it must perform. The former is called the model and the latter the function.

The work explains how the model and the function of the specific CS can be obtained by refinement of the model and function of the general CS after using a predefined set of domain-independent schema transformation operations.

To facilitate the refinement of the function part, the generic CS must be kept flexible enough. To this end, some of the ideas of the frameworks are applied when defining the CS, such as: the concepts of hot-spot, template methods and hook methods.

2.10 Ontology-Driven Information Systems: Pruning and Refactoring of Ontologies (by Jordi Conesa)

Ontology-Driven Information Systems is a method to develop conceptual schemas as refinements of the knowledge contained in general ontologies, instead of generating the conceptual schema from scratch. It differs from the previous proposal in that the starting point is not a general conceptual schema for the domain but a general ontology, which is supposed to be a *universal* conceptual schema, and thus, the initial conceptual schema is much larger. On the other hand, this proposal only addresses the static part of the IS.

The method comprises three different steps: refinement, pruning and refactoring of ontologies. During the refinement the general ontology is extended (if necessary) with the knowledge required to model the IS. After that, the ontology contains all the necessary knowledge but it is too large to be used as a final conceptual schema.

That is why we need the pruning phase. During this phase we delete from the ontology all the superfluous elements to the final conceptual schema. Afterwards, we may apply some refactoring techniques to the resulting conceptual schema to obtaining the final schema.

The main focus of the work is the development of the pruning and refactoring phases. The work describes a method that given a set of concepts of direct interest deletes all the irrelevant concepts of the extended ontology. The concepts of direct interest are those appearing in the requirements of the IS we want to specify. Taking into account this set of concepts and the relationships between them and the other elements of the ontology, the method guarantees the removal of the irrelevant elements without losing any of the needed semantics of the IS.

3 Honorary Best Student Paper Award

In conjunction with the Conference Chair of the UML 2004, the Doctoral Symposium organizers decided to promote an honorary best student paper award. The members of the program committee attending the Symposium, along with the organizers, based their judgment on the contents and actual presentation of the work. Overall participation in the workshop was also considered. Based on these criteria, Jörn Guy Süβ received the award for his work titled "A Standards-based UML-profile for message-based information dissemination". The award was sponsored by the Director of Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, and presented by Ana Moreira, the UML 2004 Conference Chair. The award consisted of a certificate and was accompanied by an attractive, fully illustrated book on Portugal.

4 Conclusions

The first Doctoral Symposium of the UML conference series can be considered a success. The participants considered the comments made by the reviewers to the submitted abstracts and the feedback received during the presentation Symposium as a positive contribution that will help them to improve the quality of their work.

The presence of a significant number of workshop attendees (over 20), including participants, committee members, organizers and several PhD students contributed to lively discussions which challenged each presenter within a very constructive environment.

Although no time was allocated explicitly from the schedule for a general discussion on common problems for PhD Students or for comparisons between different educational institutes PhD programs and working conditions, we should stress the funding problems that the students need to overcome when doing their thesis: two of the selected students and one of the organizers could not attend the Symposium due to funding problems. Most of the PhD students participating in the Symposium were not registered for the main conference. We believe that, when doing a PhD thesis, it is of the greatest importance to attend as many conferences as possible, and thus more efforts should be devoted to make this possible.

The Doctoral Symposium will continue at the Models 2005 conference, with Jeff Gray as chair handling the organization. It will have a different format, where each student will be assigned a mentor who will lead the discussion following the student's presentation. Due to the mentoring aspect of the event, the Symposium will be open only to those students and mentors participating directly in it. We expect this will improve the quality of the Symposium even further. Further details on the new edition of the Doctoral Symposium can be found at:

`http://www.cs.colostate.edu/models05/cfpDoctoralSymposium.html`

Acknowledgments

The organizers of the Symposium would like to thank all the seniors that volunteered to make the Symposium a great success. In particular, we are very grateful to all the members of our Senior Program Committee, and especially to the seniors that attended the workshop and shared some of their time helping the students to improve their work.

We would also like to thank Ambrosio Toval and Ana Moreira, for their valuable advice during the preparation of the Symposium.

Finally the organizers are grateful to all PhD students attending the Symposium for submitting and/or sharing their ideas.

Function Net Modeling with UML-RT: Experiences from an Automotive Project at BMW Group

Michael von der Beeck

BMW Group,
80788 München, Germany
Michael.Beeck@bmw.de

Abstract. This paper presents the *function net modeling* approach that has been developed within an automotive project at BMW Group aimed at software development for electronic control units. This modeling approach provides a graphical, quite abstract representation of a (typically large) set of functions to be realized in software or hardware. Function net models are described in UML-RT, a dialect of the UML. They only represent structural information, where two architectural views are precisely separated: the logical view uses capsule structure diagrams of UML-RT in order to model independent of (later) HW/SW design decisions. Design decisions are taken in the technical view using UML-RT's component and deployment diagrams. The development of function net models is tightly integrated within several activities of the overall system development process: with requirements engineering, with behaviour modeling and code generation as well as with version and configuration management.

1 Motivation

Within the development of embedded systems in the automotive industry the following challenges have to be mastered:

– The number of functions to be realized increases considerably.
– The average complexity of the functions themselves grows.
– A vast number of interactions exists between functions – even between functions from different domains e.g. the comfort and chassis domains.
– A migration from hardware (HW) to software (SW) solutions occurs: more and more functionality has to be realized in SW.
– Safety-relevant applications require high quality assurance and especially high correctness issues.

Putting these pieces together, a very complex task of system development results. In order to master this complexity a clear graphical system view of the overall set of functions is necessary. We denote models that provide such an overview as *function net models*.

Within an automotive project at BMW Group in the context of safety-relevant systems for the chassis domain, a function net modeling method - based on UML-RT, a dialect of the standard modeling notation UML - has been developed and applied. Beside the purely conceptual part of the function net modeling approach we have also developed comprehensive tool support which integrates function net modeling in a seamless system development process from requirements engineering to code generation.

The rest of the paper is structured as follows:

In section 2 the set of requirements for the function net modeling approach is presented. Section 3 describes why UML-RT has been selected as modeling language for the function net modeling approach. Modeling rules for function net modeling are discussed in section 4. The use of the UML-RT modeling constructs is presented in section 5. In section 6 we focus on the separation between logical and technical architecture. Section 7 discusses why function net modeling is (up to now) restricted to structural modeling. The tight integration of function net modeling in an overall system development process is described in section 8. The experiences with function net modeling in an automotive project at BMW Group are summarized in section 9. Finally, we draw our conclusions and make several suggestions for possible future enhancements.

2 Requirements for Function Net Modeling

In the following subsections we discuss the requirements for function net modeling: general requirements as well as requirements for method and tool support.

2.1 General Requirements

In order to achieve a clear graphical system view of the set of all (or a large number of) functions to be developed, the function net modeling method must satisfy the following requirements:

- Function net models must offer an intuitive understanding - even for non-experts reading and modifying function net models.
- Function net models must allow efficient modeling.
- The function net modeling approach should use abstraction and hierarchical structures in order to avoid overwhelming model representations.
- For clarity reasons function net models shall be developed with a precisely defined modeling notation.

2.2 Requirements on Method and Tool Support

Function nets must provide a graphical representation of functions, SW-components, HW-components, and of partitioning information. Furthermore, interfaces (e.g. function interfaces) must be precisely defined. Functions shall be modeled in a hierarchical way. Moreover, a clear distinction between logical and technical architecture modeling has to be provided. Finally, instance information (e.g. function instances) as well as type information (e.g. function types) shall be presentable. Function net models must be developed in a tightly integrated way with other development activities. Particularly, integration with requirements engineering, with behaviour modeling, and with code generation must be provided.

3 Reasons for Using UML-RT as Modeling Language

Taking into account the set of requirements for function net modeling, first of all, we intended to use the UML (Version 1.x) as modeling language due to its comprehensive set of expressive, intuitive modeling notations and because the UML constitutes an OMG standard and – more important - a quasi-industrial modeling standard. Furthermore, quite comprehensive UML tool support exists. However, UML is not tailored for a specific application domain – and especially not for embedded systems - and its application for the structuring of (embedded) systems exhibits a few insufficiencies: hierarchical modeling of structures is only supported by aggregation relationship. Furthermore, interface definitions do not distinguish between elements to be imported and those to be exported. In contrast, UML-RT, which can be regarded as a specialization of the UML for embedded systems, supports component-oriented modeling. In particular, UML-RT exhibits the following properties: structures can be defined in an evidently hierarchical way. In addition, interface elements can be distinguished between import and export elements. Comprehensive tool support (Rational Rose RealTime [1]) exists for UML-RT, such that we selected the UML-RT notation for function net modeling.[1]

4 Modeling Rules for Function Net Modeling

Using a modeling language like UML-RT does of course not guarantee that adequate function nets are developed. In addition, modeling rules are necessary which determine how the notation UML-RT should be used. Therefore, a comprehensive set of constructive and precise modeling rules for function net modeling has been developed. The set of modeling rules cannot be presented in detail in this article. However, an overview will be given in the following.

Some modeling rules restrict the use of UML-RT for function net modeling:

– Function net models only specify structural information, no behaviour information. Therefore, only structure diagrams, component diagrams, and deployment diagrams of UML-RT are used.
– Attributes and operations are not used.
– The use of ports and protocols has been restricted.

Some other modeling rules provide naming conventions:

– Naming conventions exist for capsules, capsule roles, signals, protocols, ports, connectors, components, and component instances.
– Stereotypes are used to determine specific uses of nodes for modeling busses, sensors, actuators, and gateways.

More complex rules support the development of clearly arranged function nets:

– One such rule describes the use of mediators. These are (stereotyped) capsules that gather several connectors to reduce the (visible) complexity of function nets.

[1] Nowadays, we would choose UML 2.0 as the modeling language for function net modeling, because it offers all modeling capabilities of UML-RT, but will probably constitute a more widespread standard modeling language in future.

We not only defined the modeling rules for function net modeling, but also enhanced the commercial UML-RT tool Rational Rose RealTime by analysis functions, which check whether a given UML-RT model fulfills the modeling rules.

5 Use of the UML-RT Modeling Constructs

Before explaining the use of the diverse UML-RT modeling notations, we provide a very simple function net model example by figure 1. This figure presents a system that calculates the car velocity from signals provided by four wheel sensors and which displays the resulting velocity.

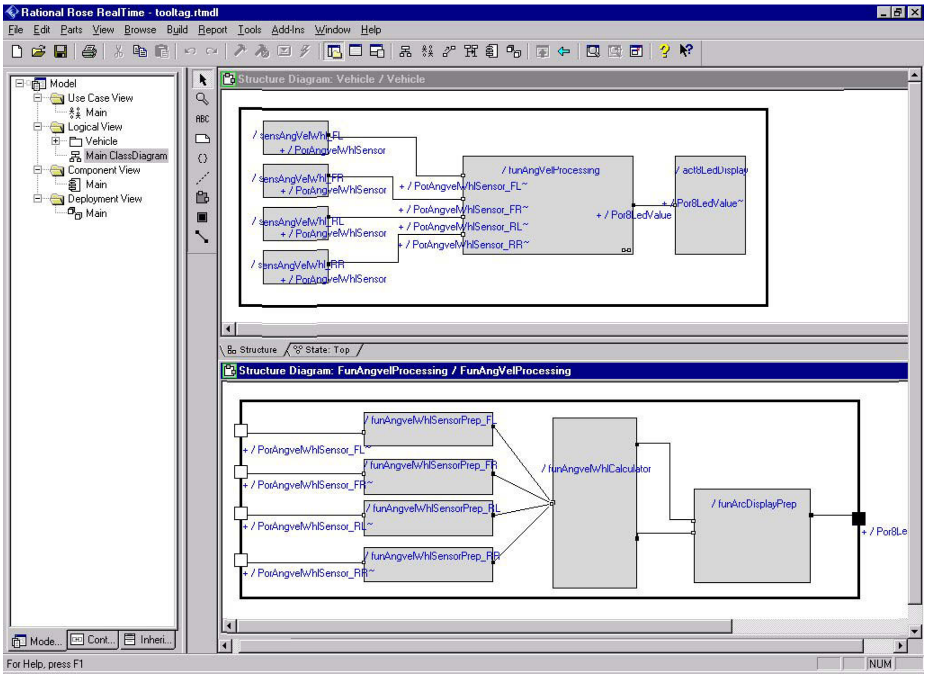

Rational Rose RealTime - tooltag.rtmdl
File Edit Parts View Browse Build Report Tools Add-Ins Window Help
Model
Use Case View
Main
Logical View
Vehicle
Main ClassDiagram
Component View
Main
Deployment View
Main
Mode... Cont... Inheri...
Structure Diagram: Vehicle / Vehicle
/ sensAngVelWhlFL
+ / PorAngVelWhlSensor
/ sensAngVelWhlFR
+ / PorAngVelWhlSensor
/ sensAngVelWhlRL
+ / PorAngVelWhlSensor
/ sensAngVelWhlRR
+ / PorAngVelWhlSensor
/ funAngVelProcessing
+ / PorAngvelWhlSensor_FL~
+ / PorAngvelWhlSensor_FR~
+ / PorAngvelWhlSensor_RL~
+ / PorAngvelWhlSensor_RR~
+ / Por8LedValue
/ act8LedDisplay
+ / Por8LedValue~
Structure
State: Top
Structure Diagram: FunAngvelProcessing / FunAngVelProcessing
/ funAngvelWhlSensorPrep_FL
/ funAngvelWhlSensorPrep_FR
/ funAngvelWhlSensorPrep_RL
/ funAngvelWhlSensorPrep_RR
+ / PorAngvelWhlSensor_FL~
+ / PorAngvelWhlSensor_FR~
+ / PorAngvelWhlSensor_RL~
+ / PorAngvelWhlSensor_RR~
/ funAngvelWhlCalculator
/ funArcDisplayPrep
+ / Por8Le
For Help, press F1
NUM

Fig. 1. A logical function net model describing the structure of a system calculating the car velocity from four wheel sensors and displaying the resulting velocity

5.1 UML-RT Language Constructs for the Logical Architecture

Capsules are used to model functions. They offer precisely defined interfaces, the communication is signal based and defined by interfaces and channels. Functions can be hierarchically refined by a system of communicating (sub)functions. Ports and protocols (of capsules) model function interfaces. A port specifies a communication point of a capsule. The protocol associated with this port contains two sets of signals: the set of signals, which can be exported at this port and the set of signals, which can be imported at this port. Connectors model channels between function interfaces.

5.2 UML-RT Language Constructs for the Technical Architecture

Components of UML-RT are used for the modeling of SW-components, whereas nodes are used to model HW-components like ECUs (Electronic Control Units), sensors, actuators, and gateways.

6 Distinction Between Logical and Technical Architecture

The function net modeling method offers a clear separation between logical and technical architecture. In the logical architecture a function is modeled without taking any information into account how this function will be realized in software or hardware, i.e. by SW-components or HW-components. However, this realization information is given in the technical architecture. Logical and technical architecture are related by the partitioning activity that describes how the logical architecture is mapped to the technical architecture. An example is given in figure 2.

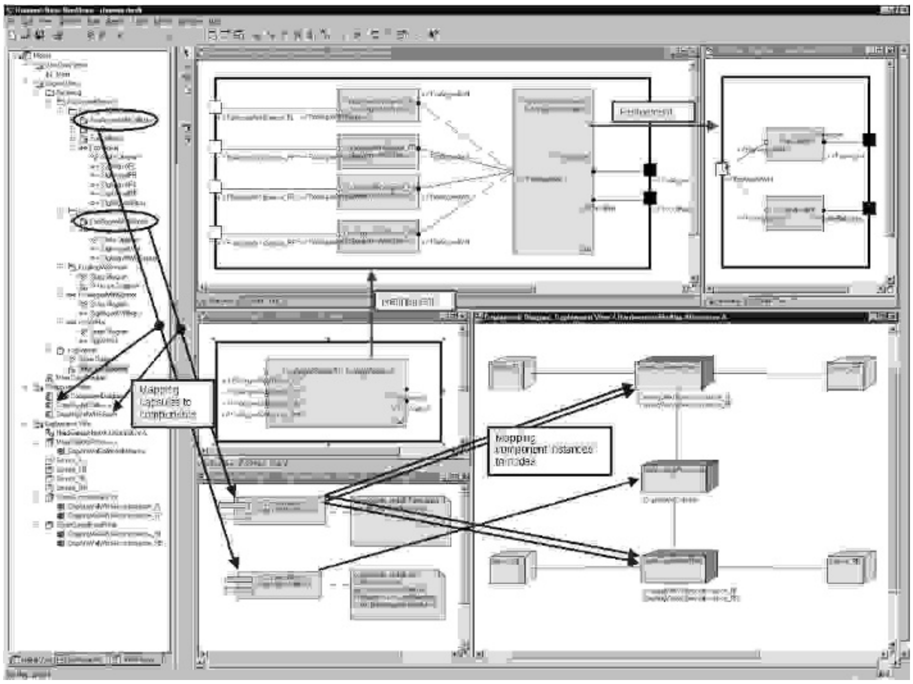

Refinement
Refinement
Mapping capsules to components
Mapping component instances to nodes

Fig. 2. A logical and a technical architecture modeled in UML-RT. Additional arrows shall illustrate mapping and refinement issues

Here, function nets for the logical as well as for the technical architecture are given. Structure diagram FunAngvelSensor shows four capsule roles of capsule type `Fun-AngvelWhlSensor` that are contained in capsule role `funAngVelSen-sorR1` – shown in structure diagram `Fahrzeug` - of type `FunAngVelSensor`.

Partitioning of these four capsule roles is now performed as follows: Capsule `FunAngvelWhlSensor` is mapped onto component `CmpAngVelWhlSensor` that is shown in a component diagram. Moreover, four instances of this component are mapped to two nodes, which are modeled in a deployment diagram: two component instances are mapped on `SignalCoordinatorFront` and the other two component instances are mapped on `SignalCoordinatorRear`.

This separation between logical and technical architecture provides several advantages:

– It simplifies the development of several (alternative) technical architectures starting from one common (logical) architecture in order to achieve optimal solutions for different quality criteria, e.g. safety-related issues (like compliance with a given safety integrity level) and performance issues (e.g. bus load optimization).
– It allows reuse of purely logical architectural information in order to develop enhanced or quite new architectures.

7 Restriction of Function Net Modeling to Structural Information

The presented function net model approach only models structural information, but no behavioural information, although UML-RT provides very appropriate modeling notations for behaviour modeling like state and sequence diagrams. The reason for this restricted use is given as follows: the use of the tool ASCET SD [2] from ETAS for code generation was mandatory in the project where the function net modeling approach was developed. Since ASCET SD models must contain complete structural as well as behavioural information before code generation begins, the question arises as to when and how the behavioural information shall be modeled in the overall development process. If behaviour is (already) modeled within function net modeling, e.g. using state diagrams of UML-RT, the resulting behaviour model information must be transformed to ASCET SD behaviour models. However, this is a non-trivial task due to the very intricate differences in the semantics of UML-RT's and ASCET SD's state diagrams. Therefore we decided to follow a pragmatic approach: to only model structural information in UML-RT and to develop a possibility for transforming UML-RT function net models (which only contain structural information) to ASCET SD models, such that behaviour modeling will only be done (subsequently) in ASCET SD.

8 Embedding Function Net Modeling in the Overall Development Process

Function net modeling comprises the development of logical as well as technical architecture. However, these development activities must be adequately embedded in the overall development process. We have realized a tight integration of function net modeling with the following activities of the overall development process:

– Requirements engineering
– Behaviour modeling and code generation
– Version and configuration management

Figure 3 illustrates these integrations. The integrations are not only defined on a conceptual level, but are also realized by comprehensive tool support. In the following subsections the integration aspects are described in more detail.

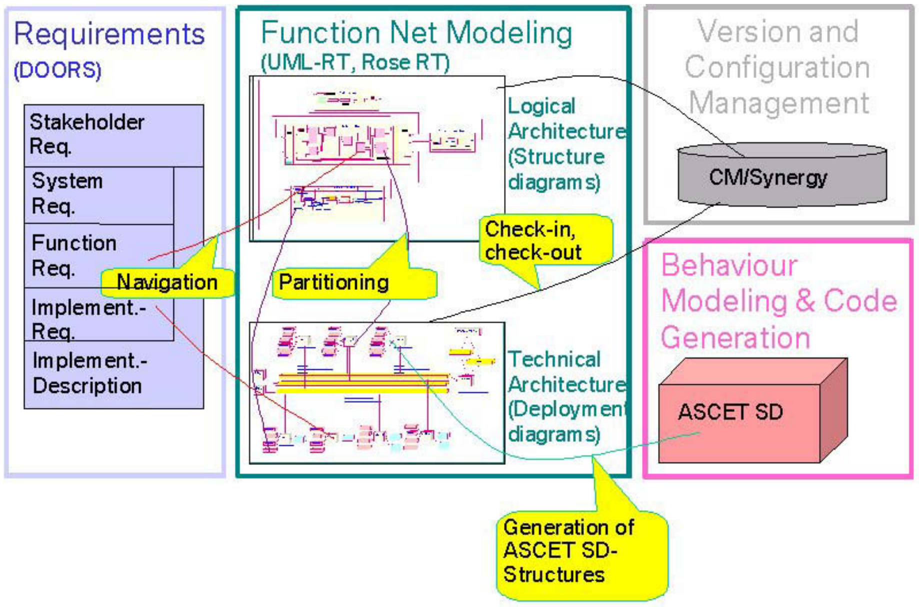


Fig. 3. Embedding of function net modeling in an overall development process

8.1 Requirements Engineering

We have developed navigational support between requirements and elements of function net models. We use the DOORS tool [3] for the modeling of requirements. For each requirement the set of function net model elements that satisfy this requirement can be shown. So if the user has selected a requirement in DOORS he can use a command which – if not already opened – opens the corresponding function net and highlights the corresponding function net model elements. This navigational support also works in the other way round.

8.2 Behaviour Modeling and Code Generation

A UML-RT function net model (which only contains structural information) has to be transformed to a (preliminary) ASCET SD model, such that this model can subsequently be manually enriched by behavioural information. Then, the resulting ASCET

SD model (containing structural as well as behavioural information) is used by the ASCET SD tool for code generation. We have developed a tool, which automatically performs the abovementioned transformation from UML-RT function net models to (equivalent) ASCET SD models.

8.3 Version and Configuration Management

In order to allow simultaneous development of a function net model by several people a systematic support of version and configuration management is necessary. Analogous to the requirement that the tool ASCET SD shall be used for code generation there has been the requirement that the tool CM Synergy [4] shall be used for version and configuration management. Since there did not exist an adequate integration between Rational Rose RealTime and CM Synergy, we developed an integration between these tools such that arbitrary parts of a UML-RT function net model can be stored separately.

9 Experiences

In this section we describe experiences with the application of function net modeling within the abovementioned automotive project at BMW Group.

9.1 Methodology Issues

The function nets, which have been modeled, are quite complex. Therefore it was necessary that the function net modeling approach allows the possibility of selecting an arbitrary level of abstraction such that an adequate level of granularity can be achieved. This requirement is fulfilled by our function net modeling approach.

A clear separation between type information (e.g. capsules, protocols, signals, and components) and instance information (e.g. capsule instances, protocol instances, signal instances, and component instances) is necessary. UML-RT fulfills all of these separation issues - except the distinction between signals and signal instances: UML-RT only offers signals (which provide type information). Therefore we developed a capability to also model signal instances in UML-RT.

As described in section 4, a comprehensive set of constructive and precise modeling rules has been developed. They simplify the development of function net models and are important to achieve a common understanding of function net models. In addition, the possibility of automatic checks, whether function nets satisfy the modeling rules, tremendously simplifies development.

Experience with object-oriented notations helps users to apply the function net approach. However, the set of UML-RT diagram notations to be used for modeling function nets is quite restricted, so that potential users of the function net modeling approach who do not have experience with object-oriented notations do not have to learn UML-RT completely.

9.2 Tool Support Issues

The possibility of extending the functionality of Rose RT by extensive use of the RRTEI (Rational Rose RealTime Extensibility Interface) allowed us to fulfill the following requirements:

– to realize automatic checks of function net models with respect to our modeling rules for function nets
– to develop a user friendly support for the partitioning process (i.e. to determine which capsule instances are mapped to which node instances)
– to automatically perform bus load estimations
– to integrate function net modeling tightly in an overall system development process.

Therefore, we could combine the use of a commercially available UML-RT tool with the realization of project specific requirements from BMW Group.

9.3 Integration Issues

In general, the tight integration of function net modeling in an overall system development process was an indispensable precondition for gaining acceptance for the function net modeling approach within the application project. In addition, we discovered the following issues:

The work to link all requirements with corresponding elements of the function net model is quite a tedious task. Furthermore, it is necessary that the requirements be structured in a systematic way such that it is easier to decide which requirements have to be linked with which function net model elements.

Tool support for the transformation from UML-RT function net models to ASCET SD models is a necessary precondition if this transformation shall be performed in an efficient way. However, our transformation tool up to now only works in a unidirectional (only from function net models to ASCET SD models, but not vice-versa) and non-incremental way (it always transforms complete function net models, such that previously generated or manually enhanced ASCET SD models are overwritten). To achieve a more efficient overall development process a bi-directional and incremental transformation tool would be necessary.

Finally, adequate tool support for versioning and configuration of (parts of) function net models is indispensable if several persons develop a common function net model.

10 Conclusions

In this paper the function net modeling approach has been presented. This approach provides a graphical, quite abstract representation of a (typically large) set of functions to be realized in the automotive domain.

A complete method has been developed:

- It is based on the given modeling notation UML-RT.
- It provides a set of constructive modeling rules which are specifically designed for the development of function nets. In particular the rules define that only a small part of the UML-RT notation is used for the development of function net models.
- The method comes with comprehensive tool support. This has been achieved by using a commercially available UML-RT tool which has been extended by individual scripts in order to provide additional functionality and which has been tightly integrated with other tools that support other activities in the overall development process.

11 Outlook

It is quite obvious that a migration from UML-RT to UML 2.0 for the modeling of function nets might be promising, since UML 2.0 will constitute a more standardized and widespread language than UML-RT. In addition, the migration work should be justifiable, because UML-RT concepts like ports, protocols, connectors, and capsules (which however do not exist in UML 1.x) are either contained in UML 2.0 or can be easily substituted by UML 2.0 modeling concepts.

Alternatively, an automotive-specific UML 2.0 profile e.g. based on the EAST-ADL [5], i.e. the architecture description language of the ITEA project EAST-EEA, or a systems engineering specific profile of UML 2.0 like SysML [6] could substitute UML-RT as the basis of function net modeling.

Our restriction of function nets to a specification of purely structural issues can be seen as a pragmatic decision in order to avoid intricate (semantic) transformations between behavioural models in UML-RT and Ascet SD. However, in principle it makes sense to explore behaviour modeling within the application domain of function nets. With respect to a possible migration to UML 2.0 the use of protocol automata for the modeling of behaviour in function interfaces could be promising, such that interfaces will no more comprise just signatures, but real behavioural entities.

Finally, the integration aspects of function net modeling could be extended. It might be promising to examine how testing activities can be tightly integrated with function net modeling.

Acknowledgement

The function net modeling approach and its tool support were developed with the help of many colleagues: in particular, I would like to thank M. Bayerlein, M. Huang, M. Köhlert, P. Mai, R. Mehlhorn, Th. Nowak, F. Ratzinger, R. Sandner, W. Schwerin, and B. Wimmer for their very noteworthy contributions.

References

1. Rational Rose RealTime: http://www.rational.com
2. ASCET SD: http://en.etasgroup.com/products/ascet_sd
3. DOORS: http://www.telelogic.com/products/doorsers/
4. CM Synergy: http://www.telelogic.com/products/synergy/
5. EAST-EEA Embedded Electronic Architecture, Deliverable D3.6, Definition of language for automotive embedded electronic architecture, http://www.east-eea.net
6. SysML (Systems Modeling Language), http://www.sysml.org

Supporting the Building and Analysis of an Infrastructure Portfolio Using UML Deployment Diagrams

Jeffrey A. Ingalsbe

Ford Motor Company
jingalsbe@ford.com

Abstract. Rational, objective analysis of an infrastructure portfolio requires that the portfolio be represented in a consistent and understandable way. For an organization whose portfolio includes legacy systems, systems under development, systems on fundamentally different and incongruous platforms, and systems with little or no documentation, the task is daunting. This paper describes work currently being undertaken to represent the IT portfolio of Ford Motor Company from an infrastructure perspective using UML deployment diagrams. The objective of the work is to support the analysis of the portfolio and its subsequent alignment with key IT strategies. To accomplish this, the UML deployment diagram was extended and a template created. This paper discusses the extensions, the template, and its ongoing deployment to the organization. Tool considerations and future work are discussed as well

1 Introduction

Alignment of IT strategies with key business strategies has been understood for some time to be a top priority of organizations interested in getting the maximum value out of their IT dollar [1]. However, optimal alignment requires the capability to rationally, objectively analyze an existing IT infrastructure portfolio [2] which, in turn, requires that the portfolio be represented in a consistent and understandable way. For an organization whose portfolio includes legacy systems, systems under development, systems on fundamentally different and incongruous platforms, and systems with little or no documentation, the task is daunting. Additionally, even if the task were undertaken, there might be organizational change constraints that would work against its success. This paper describes work currently being undertaken to represent the IT portfolio of Ford Motor Company from an infrastructure perspective using UML deployment diagrams. The objective of the work is to support the analysis of the portfolio and its subsequent alignment with key IT strategies. Specifically, those strategies are to optimize cost, quality and value to Ford through simplification. To accomplish this, the UML deployment diagram was extended and a template created. This paper discusses the extensions, the template, its initial deployment to the organization, tool considerations, and future work.

2 Scope and Context

The representation and subsequent analysis of the IT portfolio of Ford Motor Company from an infrastructure perspective are on-going and being accomplished by the execution of an alignment process. It prescribes the systematic, collaborative, building of the portfolio, its analysis to find alignment opportunities, and its transformation through the execution of those alignment opportunities. This paper focuses on the UML deployment diagram extensions and its deployment. It does not, however, address in detail the end-to-end process in which it was used. That is the topic of another paper.

3 Choosing the UML: The Risks

The Unified Modeling Language is a set of notations that provide a standard way to visualize, specify, construct, and communicate the artifacts of software intensive systems [3]. It is considered by most to be the defacto standard. UML **deployment diagrams** are used to model the static deployment of a system by showing the configuration of run time nodes and the components that live on them [3]. Considered in the abstract the UML and specifically, the UML deployment diagram, seemed to be the correct choice to represent constituent elements of an infrastructure portfolio. Considered with reference to a specific organization with its specific goals, personnel, resources and constraints led to the identification of four risks to the success of the initiative.

3.1 Personnel (Current Workload, Training, and Competence)

The personnel posessing the knowledge required to render the deployment diagrams were not trained in the UML, could not be considered competent in the UML, and had little time to dedicate to the task. The introduction of a new notation (the UML) and a new artifact (the deployment diagram) into an organization for representing all future work is one thing. But their introduction into an organization for representing all work **past**, **present**, and **future** is quite another. While systems currently under development are likely to have dedicated architects whose methodology could be modified to include a new notation and artifact, legacy systems (i.e. systems in production) are not. After a system is released to production a different set of personnel with a much different skill set manage its operation and maintenance. Additionally, those personnel may manage multiple systems, not just one. Consequently, the risks were: the personnel had no formal training (formal training would cost and take time), could not be considered competent (becoming competent would cost and take time), and had existing workloads (the impact to productivity while being trained, becoming competent, and then doing the work would be a reason to resist the initiative).

3.2 Tools (Cost and Deployment)

Object modelling tools were expensive and the time required to evaluate, select, purchase, and deploy would introduce an unacceptable delay. Object Modeling tools that support the UML are costly. Evaluation, selection, purchase, and deployment of a tool to support the composition of UML deployment diagrams by application owners numbering in the hundreds would be costly and time consuming. The risk: without a modeling tool, could the deployment diagrams be built?

3.3 Knowledge Base (Sample Libraries and Similar Efforts)

In this paper, a knowledge base is used to mean a collection of samples, rules of thumb, white papers, and lessons learned related to producing UML deployment diagrams for an enterprise. To the best of our knowledge, the knowledge base for deployment diagrams is small as compared to the other UML models. Perusing the knowledge bases for the major tool vendors reveals that the use of deployment diagrams has not progressed to the point where there is a lot of samples or knowledge. Therefore, the effort to build an infrastructure portfolio based on the UML deployment diagram would have to be joined with an effort to build a library of deployment diagrams for different classes of systems. E.g. data warehouse systems, web services systems, traditional client server systems, etcetera. The risk was: without a knowledge base from which the application owners could draw, could the deployment diagrams be built?

3.4 Usage (Across Business Units at Multiple Levels)

A conceptually accessible deployment diagram to be used across business units by technical personnel for analysis, by middle management for planning, and by senior management for understanding might be impossible to create. The infrastructure portfolio's use would range vertically from technical personnel through middle management and horizontally across business units. Therefore it had to be accessible conceptually across a wide range of personnel. The risk was: with such a diverse user base, could a useful deployment diagram be produced?

4 Choosing the UML: Mitigating the Risks

The following subsections detail the decisions that were made to mitigate the risks of choosing the UML.

4.1 Regarding Personnel (Current Workload, Training, and Competence)

The knowledge of the content of the deployment diagrams was distributed. The competency to render them using the UML was central. A S.W.A.T. team was formed to produce a template (to ensure consistency and understandability) and review drafts as they were produced. The goal of this effort was to represent the portfolio in a consistent and understandable manner such that it could be rationally and objectively analyzed. Since the current workload of the application owners and architects was already significant, and the cost of training all of them was too high, and the time lapse required for all of them to gain competency was too long even if training was affordable, it seemed clear that some of the effort and all of the competence would have to be provided centrally. That is, a core team would be created to provide the competence in building UML deployment diagrams but the application owners and architects would still have to gather and provide the data necessary to render the diagrams. The S.W.A.T. team would be trained and become competent in creating deployment diagrams using the UML This S.W.A.T. team would create a template to be used as a starting point by the application owners and architects (containing all the information necessary to conduct the analysis of the portfolio). The application owners would then use the template to build the draft deployment diagrams and would then review those drafts with the S.W.A.T. team for sign off.

4.2 Regarding Tools (Cost and Deployment)

The effort had to begin immediately, but the tool would have to go through the evaluation, selection, purchase, and deployment processes which would introduce an unacceptable delay. An office automation tool was specified for tactical use while the strategic tool wound its way through the corporate processes. Implicit in the goal to represent the portfolio in a consistent and understandable way such that it could be rationally and objectively analyzed was that the analysis take place in a **timely manner**. That is, the analysis was intended to support the optimization of cost, quality and value to Ford through simplification of the infrastructure portfolio. That optimization could not wait several quarters to begin. Since the cost of object modeling tools was high and the time to evaluate, select, purchase, and deploy the was long, neither could the tool be provided for all application owners and architects, nor could the optimization wait for the tool to be evaluated, selected, purchased, and deployed. Therefore, it seemed clear that an interim tool would have to be used while waiting for the final tool to wind its way through the process and be used by the S.W.A.T. team. Diagrams rendered in the interim tool would have to be transferred into the final tool upon its arrival. Therefore, discussions with tool vendors would have to involve the automation of this process.

4.3 Regarding Knowledge Base (Sample Libraries and Similar Efforts)

To the best of our knowledge, a usable knowledge base related to the production of deployment diagrams for the enterprise did not exist. The S.W.A.T. team (see above) would produce the knowledge base in parallel with the effort by identifying classes of systems and lessons learned by those pesonnel leading the effort. Providing a template to be used as a starting point by the application owners and architects (containing all the information necessary to conduct the analysis of the portfolio) would help ensure consistency and understandability but would not capitalize on the fact that across all business units there are classes of systems whose deployment would be remarkably similar (e.g. E.g. data warehouse systems, web services systems, traditional client server systems, etcetera). Therefore, it seemed clear that knowledge base would have to be created which would grow as the documentation of the portfolio grew. The S.W.A.T. team would identify classes of systems as they were encountered and enter them into the knowledge base.

4.4 Regarding Usage (Across Business Units at Multiple Levels)

The template (see above) would be specified collaboratively with concerned parties from each class of user. Thus ensuring maximum understandability horizontally and vertically in the organization. Since Ford IT is a huge global enterprise spanning multiple brands, multiple business units, multiple continents, and multiple packages of management it would be seem that developing a deployment diagram that would be conceptually accessible (understandable) to personnel vertically and horizontally within the organization would be impossible. It seemed clear that the template would only be successful if developed in collaboration with key concerned parties vertically and horizontally within the organization.

5 Extending the UML

The rich set of modeling concepts and notations that are provided by the UML are sufficient for many software modeling projects. In order to meet the requirements of this effort it was necessary to make use of the three built-in extension mechanisms : Constraints, Stereotypes, and Tagged Values. These three extension mechanisms are used separately and together to define new modeling elements that have distinct semantics, characteristics, and notation relative to the built in UML modeling elements specified by the UML metamodel. The following sections discuss how each extension mechanism is used in the template.

5.1 Stereotypes

Stereotypes represent a subclass of an existing metamodel element with the same form (attributes and relationships) but with a different *intent* [UML 1.3]. Generally stereotypes represent a *usage* distinction. They may have required tagged values that add information needed by elements with the stereotype. In the template, stereotypes are used to create new model elements from packages and nodes. That is, package and node elements are stereotyped to allow them to have additional meaning as used in the context of our template. It could be argued that stereotyping the packages may have not been necessary because a named package with a constraint on its contents and a diagrammatic convention for displaying it would have sufficed. However, it seemed conceptually consistent to stereotype the package.

5.2 Tagged Values

A *tagged value* is a keyword-value pair that may be attached to any kind of model element (including diagram elements as well as semantic model elements). The keyword is called a *tag*. Each tag represents a particular kind of property applicable to one or many kinds of model elements. In the template, specific tagged values are attached to stereotyped nodes.

5.3 Constraints

A *constraint* is a semantic relationship among model elements that specifies conditions and propositions that must be maintained as true; otherwise, the system described by the model is invalid (with consequences that are outside the scope of UML). Certain kinds of constraints (such as an association “xor” constraint) are predefined in UML, others may be user-defined. A constraint represents semantic information attached to a model element, not just to a view of it. A constraint is shown as a text string in braces ({ }). In the template, constraints are used to restrict the contents of packages, restrict association properties, and restrict tagged values on nodes.

6 The Template

The Template is a set of rules and guidelines (enforced by UML extensions) that when constructed yields a deployment diagram. It could have been considered a top level package or, possibly a stereotype of a diagram. Diagrammatically, the adornments for components were suppressed to help keep the diagrams to one page, although there is no serious impact if they are not. The UML template was developed iteratively and collaboratively with input from information architects, application architects, security architects, infrastructure architects, operations personnel, middle management personnel, senior management personnel, capacity demand management personnel, and optimization consultants. At each and every juncture the template was evaluated with respect to four goals:

6.1 Consistency

To support analysis across a group of deployment diagrams certain things about individual deployment diagrams must hold true, including a standard style (fonts,

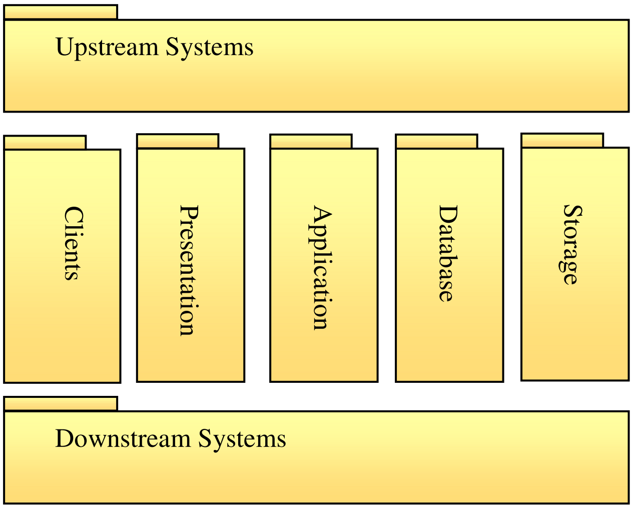


Fig. 6.1. Real estate apportionment

adornments, use of color), a standard real estate apportionment (division of the model real estate into regions which are assigned a specific purpose), and consistent specification of technology products. Figure 6.1 shows how the real estate of the template was apportioned using packages. Each package has constraints on its position relative to the other packages in the model and on the nodes it can contain.

Understandability: To support usage by diverse groups with varying intentions certain things about individual deployment diagrams must hold true, including naming of model elements (nodes, packages, stereotypes, constraints, and tagged values) that is understood by all concerned parties.

Support of analysis: It was understood from the outset that the portfolio would be initially analyzed relative to a handful of known drivers, including utilization, end of life, end of lease, cost, kinds and versions of equipment, and kinds and versions of technologies. To support that analysis across groups of deployment diagrams certain information must be present on individual deployment diagrams, including financial data, utilization data, and technology product data.

Support of resultant actions: It was also understood from the outset that certain actions would be called for as a result of the analysis, including consolidation on like technologies or nodes, migration to new or shared technologies or nodes and decommissioning of technologies or nodes. To support those actions across groups of deployment diagrams certain information must be present on individual deployment diagrams, including things like interrelationships between systems (upstream and downstream systems).

The result of the collaborative iterations was a template with a common style sheet, real estate apportionment using stereotyped packages with constraints, consistent naming of model elements, and stereotyped nodes.

6.2 Real Estate Apportionment Packages

Supporting the goal of consistency required a standard real estate apportionment. That is, it needed to be clear when viewing one of hundreds (perhaps thousands) of deployment diagrams that certain groups of elements would always be found in a certain area of the model. For example, upstream systems (other systems that master data which is used by the system under question) always appear in a package located on the top of the model whose width spans the entire model. Accomplishing this required stereotyping the packages, placing constraints on the position of each package relative to the other packages and placing constraints on the contents of the packages. The following subsections present all of these.

6.3 Technology Products and Services

Supporting the goal of consistency required the consistent specification of technology products in the models. That is, a line drawn between an application node and a database node indicates a communication association exists. The properties of that association might indicate that the **network services** are provided by the LAN and IP Services and protocols. While the **data access services** are provided by sql net. Understanding the services provided by a technology product requires referencing the Ford Architecture Framework which provides a structure for classifying and organizing the components of the Ford Enterprise Architecture. Supporting the goal of consistency then required that all properties of associations and all tagged values of nodes reference the Ford Architecture Framework. If a technology product has not been classified it must be submitted to the Architecture Management group for their future consideration.

6.4 Upstream Systems and Downstream Systems

An upstream system is one that sources data for the system being modeled. A downstream system is one that sinks data for the system being modeled. They are represented as stereotyped nodes. An upstream system could be a directory service that masters user identification data used by the system to determine whether a user is part of a certain organization. Access to the directory service might be on demand via a system call or nightly via an ftp pull. A downstream system could be an analytical system that receives test data from the system being modeled. Access to the analytical system might be via an ftp push of a flat file. To specify this information six tagged values were identified for upstream and downstream systems: who, what, source, target, mechanism, and frequency.

Who: This specifies the name of the system that will be sourcing the data.
What: This specifies the business data being sourced. It is a natural language description of the data, for example "car sales volume data".
Source(From): This specifies the source of the data in general terms, for example"db2 tables on Facility C".
Target(To): This specifies the target of the data, in general terms, for example "a flat file on facility C".
Mechanism: This specifies the mechanism for the data transfer between the source and the target, for example "via an ftp pull or push" or "via an sql query".
Frequency: This specifies the frequency of the data transfer between the source and the target, for example <u>D</u>aily, <u>W</u>eekly, <u>M</u>onthly, or <u>O</u>n-Demand. daily More detail can be added, for example D2x would specify that the transfer takes place twice daily.

When completely specified, the tagged values of an upstream system can be strung together to form a sentence that makes sense. E.G. Who provides what from this source to that target via this mechanism at that frequency. Using a concrete example: System X provides user training data from an Oracle database to a flat file via an ftp push, nightly.

Each node represents one "kind of" upstream system. For example, if there are 17 upstream systems sourcing the same data to the system in the exact same format then one may draw one node and specify the 17 unique upstream systems in a legend at the bottom of the page or in an excel attachment. Please note that these upstream systems are not solely those systems that give your application feeds; they are also applications that source data in a real-time basis through mechanisms such as sql queries.

6.5 Client Package

Clients are nodes that allow actors to interact with the system being modeled. It is important to note that the actors are people not systems. Systems that interact with the system being modeled are either upstream or downstream systems. A client could be a customer interacting over the internet using a standard web browser. Alternatively, a client could be an engineer interacting over the intranet using custom written client-server software. To specify this information three tagged values were identified for upstream and downstream systems: who, kind, and software stack.

Who: This specifies the business role of the client accessing the system.
Mechanism: This specifies the mechanism used for accessing the system, for example "public internet client", "intranet client", or "3270 terminal emulator client".
Stack: This specifies the software stack or load required for the user to access the system, for example "Internet Explorer", "Citrix Client", or "Business Objects Client".

6.6 Presentation Package

Presentation nodes are those responsible for presenting the application to clients. For a typical web application, presentation nodes comprise web server components. However, application servers can function as http servers (read web servers) in addition to running application code in the context of some higher level API (read servlets, EJB, and ASP for example). So it is possible for a node to exist in the application package and the presentation package. For a typical mainframe application, there is no distinct presentation node. The mainframe presents the application via green screens. For a typical UNIX x-windows application, there is again no distinct presentation node. The application node presents the application. When load balancing is required, it may be depicted by representing each node separately and representing the load balancer as its own node. It could be argued that an xor constraint on multiple associations between a client and the load balanced servers would depict the same relationship but it seemed conceptually less accessible. The associations connecting clients to presentation nodes and presentation nodes to application nodes, for example, must specify the network services and data access services that are required (referencing the architecture framework). To specify a presentation node four tagged values were identified: machine name, software stack, security services, and monthly costs.

Machine Name: This specifies the server and its known attributes. This is linked to an asset management repository to eliminate typos.
Software Stack: This specifies the software that must be on the node in order for the application to run. It is not the entire load but the incremental load for the application being modeled.

Security: This specifies the list of the security services that the application uses, for example WSL, WSL-X, or RACF.
Monthly Costs: This specifies the monthly application costs and the monthly server costs incurred by the application running on the server.

6.7 Application Package

Application nodes are those responsible for the execution of business logic. For a typical web application, application nodes might comprise application server components. For a typical mainframe application, application nodes might comprise custom code written in Cobol executing on a mainframe. To specify an application node, four tagged values were identified: machine name, software stack, security services, and monthly costs. These are detailed in section 6.5.

6.8 Database Package

Database nodes are those responsible for database management. For a typical web application, database nodes might comprise Oracle database components on a UNIX box. For typical mainframe applications, database nodes might comprise db2 database components on a mainframe. Fail over database servers should be noted. To specify a database node, four tagged values were identified: machine name, software stack, security services, and monthly costs. These are detailed in section 6.5.

6.9 Storage Package

Storage nodes are those responsible for storage management. For a typical web application, storage nodes might comprise storage components on a SAN. For a typical mainframe application, storage nodes might comprise storage components on a SAN as well. Mirrored storage and backup storage should be specified. To specify a storage node, four tagged values were identified: function (primary, mirrored, or backup), type (local or SAN), amount (used versus allocated) and monthly costs.

Function: This specifies whether the storage is primary, mirrored, or backup.
Type: This specifies whether the storage is local or SAN.
Allocated: This specifies how much storage has been allocated.
Used: This specifies how much of the storage is in use.
Monthly Cost: This specifies whether the storage costs per gigabyte.

7 Deployment of the Template

7.1 The Pilot

The deployment of the template took place in a pilot alignment process limited to one |sub-business unit. The scope of the effort included 183 applications managed by 26 application owners. At a high level, the process involved 4 steps. Figure 7.1 shows the steps.

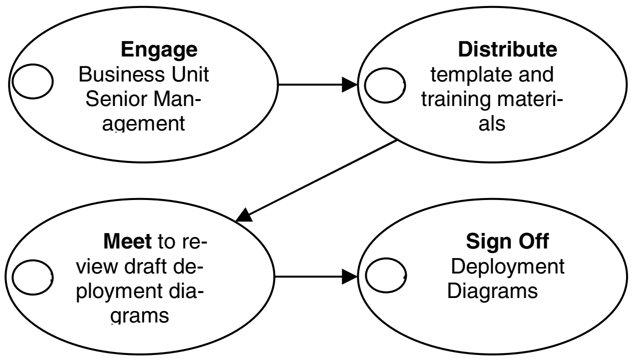

Engage Business Unit Senior Management
Distribute template and training materials
Meet to review draft deployment diagrams
Sign Off Deployment Diagrams

Fig. 7.1. High Level Pilot Process

7.2 Pilot Results

The pilot was considered a demonstrable success. Key lessons learned are documented in the following subsections.

7.2.1 Centralized Help for the Most Complex Models

The UML deployment diagram template was sent out to all application owners in the pilot. They were instructed to use the template as a starting point for rendering deployment diagrams for all of their applications. After a two week time period a review was scheduled with the S.W.A.T. team and the application owner to review each deployment diagram. In virtually every case, the application owner came to the review meeting without having completed any of the deployment diagrams, with a handful of questions and a high level of frustration. After a handful of meetings went this route, a process change was suggested and implemented. After distributing the template and training materials, a meeting would be set up with the application owners, S.W.A.T. team members, and appropriate architects. During this meeting S.W.A.T. team members would lead the collaborative building of a deployment diagram for the system that the application owner viewed as his or her most complex. The application owner and architect would then be responsible for composing deployment diagrams for the rest of the systems under their purview. After implementing this process change, virtually all application owners came to the review meeting having completed the deployment diagrams, with very few questions, and a high level of satisfaction.

7.2.2 Side Effect of Building Models

The feedback from virtually all application owners was a sense of surprise and delight that the process of building the deployment diagrams gave them an understanding of their systems that they had previously not possessed. This was especially true for application owners of legacy applications they had inherited.

7.2.3 Critical Mass

The first application owners to execute the process and build their deployment diagrams experienced more difficulty and higher levels of frustration than their peers who executed the process later in the pilot. This can be attributed to several things. First, the S.W.A.T. team was inexperienced… their skill mixture was diverse and leveled out as time went on…

7.3 Subsequent Analysis of the Portfolio

The analysis of the portfolio has begun. In the first sub-business unit to begin, analysis was performed by senior architects, and infrastructure representatives. It was performed in semi-weekly collaboration meetings where the deployment diagrams were examined as a group in order to determine whether the infrastructure supporting the deployment was underutilized, at the end of its useful life, at the end of a lease, or a nonstandard version. The result of this analysis was the identification of $5,000,000 in yearly savings.

The deployment of the template to several more business units has taken place with nearly 700 deployment diagrams having been produced and hundreds more being worked on.

8 Conclusions and Further Work

This paper described extensions to UML deployment diagrams and a template used as a starting point and guide for the creation of those deployment diagrams. The extensions and template have proven useful in the creation of consistent, understandable models the can be rationally and objectively analyzed in large numbers with respect to predefined goals like simplification and optimization of cost. Specifically, with only a fraction of the portfolio composed and analyzed , the return on investment for the effort is very high.

The template was used successfully in an environment where formal training and demonstrated competency in the UML did not exist. However, it required the efforts of a central group to collaborate on the composition of the most complex models.

Tool selection is complete although there remains work to be done to support automated analysis across models. As the number of completed deployment diagrams approaches 1,000 it is becoming imperative that these models be under formal configuration management in a searchable repository supporting analysis and interconnections between other enterprise tools. Work is ongoing in this area.

As the deployment diagrams are used more widely it is becoming clear that different organizations have knowledge of different pieces of data within the models. That is, there may exist layers of information that are viewed by different people at different times. Specifically, application architects might understand the how the components should be deployed and their interrelationships but they don't know which specific nodes those will be. Farm personnel might understand how the standard components of their farm can be home to portions of the application but they don't know how the network infrastructure operates. Work is currently being done to determine whether there should be different views of the deployment diagrams that relate to functions within the organization.

Admittedly, there are other modeling languages that could have been used and other models that could have been produced that would have allowed some of the same analysis capabilities. However, the effort would have had no synergy with other pockets of UML use within the organization, in particular in the area of systems development.

8.1 Applicability to Others

This template and process were used in the automotive domain. However, it seems clear that those with similar goals (i.e. consistent, understandable representation of an infrastructure portfolio and rational, objective analysis of that infrastructure portfolio) in different domains could find success. The template can be used universally to all types of architectures. Its usefulness in documenting legacy systems, in particular, is significant. For those looking for efficiencies, its support of analysis is perhaps its biggest benefit. However, it is believed that taking this process and template to another domain would hinge on the use of a S.W.A.T. team similar to this effort.

Finally, The template could be made more prescriptive and used for the development of future state diagrams that conform to patterns (e.g. transaction pattern, collaborative pattern, and analysis pattern).

References

[1] Dominic Barrow, "An IS Strategy We Can All Understand”, Strategy Magazine, July 2000

[2] Louis C.K. Ma, Janice M. Burn, Robert D. Galliers, Philip Powell, “Successful Management of Information Technology: A Strategic Alignment Perspective”, Proceedings of the 31st Hawaii International Conference on System Sciences, IEEE Computer Society, 1998.

[3] Ivar Jacobson, Grady Booch, James Rumbaugh, "The Unified Modeling Language User Guide”, Addison-Wesley, 1999

Model-Driven Development of Enterprise Applications

Vinay Kulkarni and Sreedhar Reddy

Tata Research Development and Design Centre, Pune, India
{vinay.vkulkarni, sreedhar.reddy}@tcs.com

Abstract. Modern business systems need to cater to rapidly evolving business requirements in an ever-shrinking window of opportunity. Modern business systems also need to keep pace with rapid advances in technology. For developing large and complex applications, industrial practice has traditionally used a combination of non-formal notations and methods. Different notations are used to specify the properties of different aspects of an application and these specifications are transformed into their corresponding implementations through the steps of a development process. The development process relies heavily on manual verification to ensure the different pieces integrate into a consistent whole. This is an expensive and error-prone process demanding large teams with broad-ranging expertise in business domain, architecture and technology platforms. We present a model-driven development approach that addresses this problem by providing a set of modeling notations for specifying different layers of a system namely user interface, application functionality and database; a set of code generators that transform these models into platform-specific implementations; an abstraction for organizing application specification into work-units and an associated tool-assisted development process. Models, being at a higher level of abstraction, are easier to understand and verify for properties of interest. Model based code generation incorporating proven design and architectural patterns results in significant gains in productivity and uniformly high quality. Models defined using these different notations are instances of a single meta model. This enables well-formedness constraints to be specified between different models ensuring their consistency leading to smooth integration of implementations of these models. The approach has been used extensively to construct medium and large-scale enterprise applications resulting in improved productivity, better quality and platform independence. We also discuss issues that need to be addressed for the approach to gain wider acceptance in the industry.

1 Introduction

Faced with the problem of developing large and complex applications, industrial practice uses a combination of non-formal notations and methods. Different notations are used to specify the properties of different aspects of an application and these specifications are transformed into their corresponding implementations through the steps of a development process. The development process relies heavily on manual verification to ensure the different pieces integrate into a consistent whole. This is an expensive and error-prone process demanding large teams with broad-ranging expertise in business domain, architecture and technology platforms. In this paper, we present a model-driven development approach that addresses this problem by providing a set of modeling notations for specifying different layers of a system namely user interface, application functionality and database; a set of code generators that transform these models into platform-specific implementations; an abstraction for organizing application specification into work-units and an associated tool-assisted development process.

Industry practice addresses scale and complexity by breaking down the problem along different axes, for instance, architecture layers and development phases as shown in Fig. 1. Functional break up results in various components. For a layered architecture the application is split up so that each piece implements the solution corresponding to a layer in the architecture. Different phases of the development process determine the properties of the application that are to be implemented during a particular phase. For example, a banking system may be broken down into different functional components like *Foreign Exchange*, *Retail banking* etc. A functional component like *Retail banking* will have a *User Interface layer* describing the way a user interacts with the system, an *Application layer* implementing the business functionality and a *Database layer* making information persistent. A development process consisting of phases such as *Analysis*, *Design* and *Implementation* will implement different properties of a layer. The Analysis phase for the application layer of *Retail Banking* will define the domain object models, use-cases etc. The Design phase will define implementation architecture and identify various design strategies such as object-relation mapping, concurrency management, auditing strategy etc. The Implementation phase will code application logic, database queries etc in the chosen platform.

Analysis	UI prototype	UML diagrams	
Design	GUI standards	Design Strategies	ER diagrams + Table design
Coding	JSP implementation	C++/Java code	RDBMS Implementation
	User Interface(UI)	**Application**	**Database**

Fig. 1. Break up of application based on development phases and architecture layers

2 Model Driven Development

The development of an application starts with an abstract specification that is to be transformed into a concrete implementation on a target architecture [2]. The target architecture is usually layered with each layer representing one view of the system e.g. Graphical User Interface (GUI) layer, application logic layer and database layer.

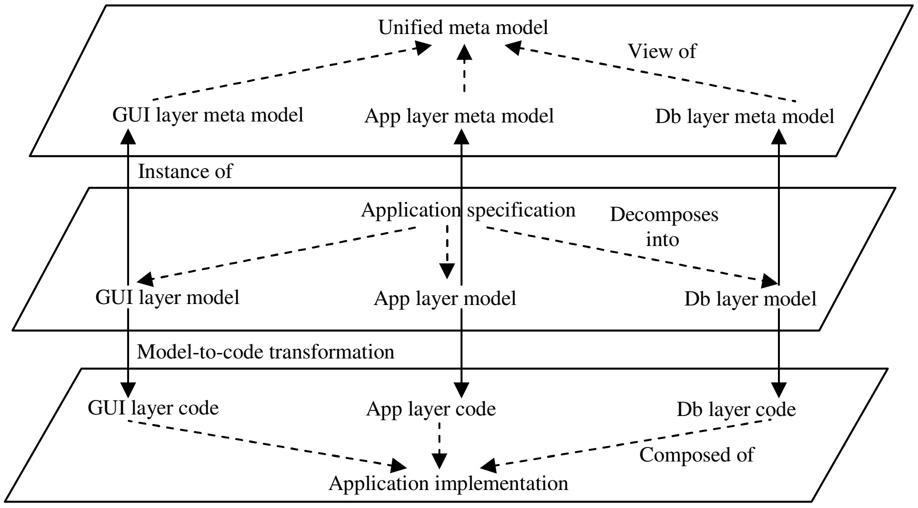


Fig. 2. Model based development approach

The modeling approach constructs the *Application specification* using different abstract views - *GUI layer model*, *App layer model* and *Db layer model* each defining a set of properties corresponding to the layer it models as shown in Fig. 2. Corresponding to these specifications are the three meta models - *GUI layer meta model*, *App layer meta model* and *Db layer meta model* which are views of a single *Unified meta model*. Having a single meta model allows us to specify integrity constraints to be satisfied by the instances of related model elements within and across different layers. This enables independent transformation of *GUI layer model*, *App layer model* and *DB layer model* into their corresponding implementations namely *GUI layer code*, *App layer code* and *Db layer code*. These transformations can be performed either manually or using code generators. The transformations are specified at meta model level and hence are applicable for all model instances. If each individual transformation implements the corresponding specification and its relationships with other specifications correctly then the resulting implementations will glue together giving a consistent implementation of the specification as depicted in Fig. 2. Models can be kept independent of implementation technology and the application specifications can be targeted to multiple technology platforms through model-based code generation. Construction of application specification in terms of independent models helps divide and conquer. Automated code generation results in higher productivity and uniformly high quality. Modeling helps in early detection of errors in application development cycle. Associated with every model are a set of rules and constraints that define validity of its instances. These rules and constraints could include rules for type checking and for consistency between specifications of different layers.

3 Architecture of a Client-Server Application

A typical client-server application is implemented across three architecture layers – user interface, application functionality and database. Each layer is implemented on a different platform supporting different primitives. For example, User interface platforms like Visual Basic provide windows and controls as implementation primitives. Application logic is implemented in a programming language like C++ or Java with classes and methods as the primitives while the database layer is implemented using tables and columns in a relational database system. These three layers are implemented independently and combined later to get an integrated implementation.

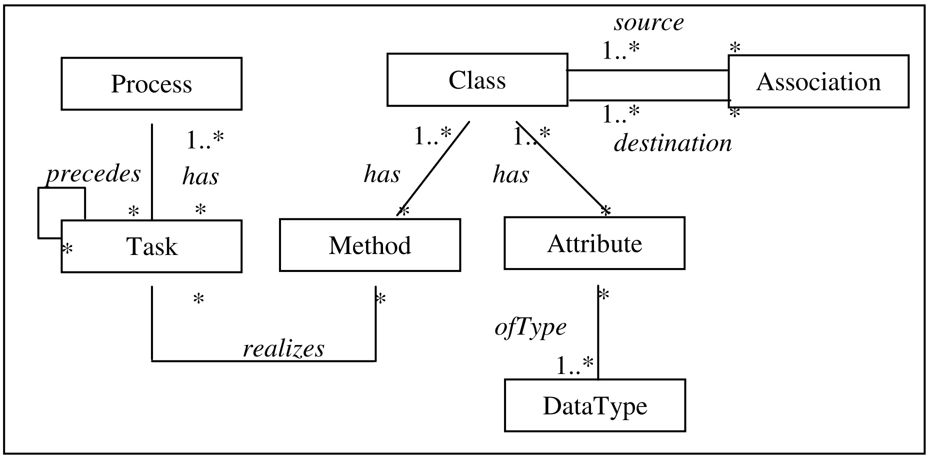


Fig. 3. Meta model for Application layer

The following sections describe the models for the different layers – Application, User Interface and Database - of a client server application in greater detail. For brevity, we have left out many details from each of the models. We also briefly describe how business logic, architectural choices and design strategies are specified.

3.1 Application Layer

The application layer implements the business functionality in terms of business logic, business rules and business process. The functionality is modeled using classes, attributes, methods and associations between classes. A business process models the application as a set of tasks and defines the order in which these tasks are executed.

Each task is implemented by a method of a class. This layer can be specified as an instance of the meta model in Fig. 3. Business logic is coded in a high level language and translated into a programming language of choice with code fragments corresponding to the selected design and architectural strategies suitably woven in.

Example: A banking system allows a user to open and operate an account with a bank. Two classes corresponding to this system are *User* and *Account*. An association between *User* and *Account* specifies the account belonging to a user. *Account number* is an attribute of *Account* and *name* is an attribute of *User*. The account opening process involves filling up an account opening form, verification of the form and approval of the form. A user can operate the account only after it is approved.

3.2 User Interface Layer

A user interacts with an application through its user interface. The user feeds in information using forms and browses over available information using queries and reports. Forms, queries and reports are implemented in a target platform using standard graphical user interface primitives such as windows, controls and buttons. A window is a unit of interaction between the user and the system and is composed of controls and buttons. A control accepts or presents data in a specific format. The user can perform a specific task by clicking on a button.

Example: The user of a banking system needs a query window to inquire about past transactions and the current balance. She also needs a form window to withdraw money from her account. These windows will use appropriate controls to represent account number and date of transaction.

The user interface is best specified in terms of windows, data to be shown in each window, controls to be used to represent this data, possible navigation between windows and actions that can be performed. The core of a model for such a specification is as shown in Fig 4. In the figure a *UIClass* represents a logical grouping of the data to be shown in a window. Each *UIClass* represents a view of the application data. The association *mapsto* between *UIAttribute* and *Attribute* defines the view. This enables type-correct representation of value of the *Attribute* on the

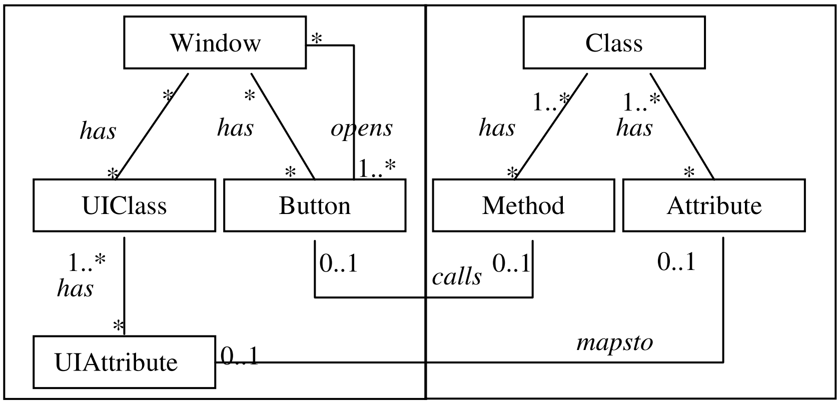


Fig. 4. Model for User interface layer

Window. This also ensures the user can enter only values that are valid for the attribute of a class. The association *calls* between *Button* and *Operation* enables type-correct invocation of operation. This also ensures the right set of objects get created and passed as parameters to the method invocation. The *mapsto* association enables copying of the right values from window to the parameter objects.

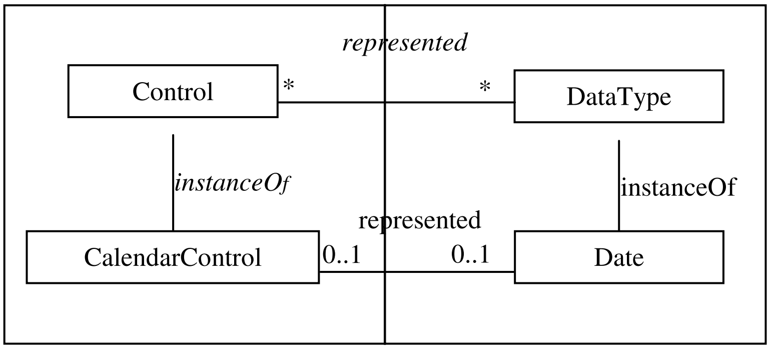


Fig. 5. A model for specifying GUI standards

Additionally for uniform look and feel of the user interface, a particular data type should be represented by the same control in all the windows. In the banking system the same format should be used for all dates in all the windows. Similarly the same format should be used for account number in all the windows. An association between data type and control, as shown in Fig, 5, will allow specification of such GUI standards.

3.3 Database Layer

The database layer provides persistence for application objects using RDBMS tables, primary key and query based access to these tables, and an object oriented view of these accesses to the application layer.

In a relational database, the schema is made up of tables, columns and keys where a column has a name and a simple data type, and relations between tables are specified using foreign keys. An object model specifies similar information in terms of classes, attributes and associations. A row in a table contains data for an instance of a class. Therefore, the mappings essential to object / relational integration are between a table and a class, between a column and an attribute, and between an association and a key as shown in Fig 6.

Example: The persistent information for the banking system will include details about accounts and users. Two tables User and Account implement this persistent information. These tables have columns corresponding to user name and account number. The association between a user and an account is implemented by having account number as a foreign key in the User table and a primary key in the Account table.

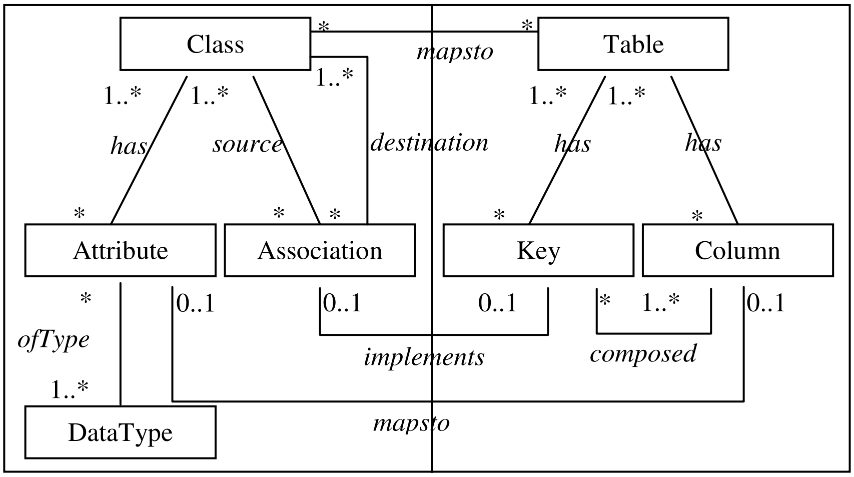


Fig. 6. Meta model for database layer

Similar to the *mapsto* association of User Interface model, the *mapsto* association between Attribute and Column ensures type correctness. The *implements* association allows correct coding of class associations using appropriate Primary and Foreign keys. This association uniquely identifies the related classes and the tables. These classes and tables must be related through *mapsto* association. Such constraints can be specified in the meta model.

3.4 Business Logic

Business logic specifies the computations to be performed by the application in terms of methods. The language for specifying business logic frees the application developer from low-level implementation concerns such as memory management, pointers, resource management etc and is retargettable to programming languages of choice such as Java, C++, C# etc.

4 Model Validation

Modeling helps early detection of errors in application development cycle. Associated with every model are a set of rules and constraints that define its valid instances. They include rules for type checking and for consistency between models of different layers. Below are a few of the validation rules for the models presented so far:

– User interface should allow specification of all *Tasks* in the business process and the interaction sequence be consistent with the *precedes* relationship between the *Tasks*
– User interface should display data that is consistent with respect to the parameters being passed to the operations invoked from the *Window*

– Database layer should ensure that *implements* association is implemented in a consistent manner. For example, the 1:M association between classes User and Account should be implemented by making the primary key of User table as foreign key of Account table.

5 Integration

Figure 7 shows specifications for a banking system. The association *has* between class User and class Account is implemented in the database layer by the column *AccNo* that is the Primary key in *Account* table and Foreign key in *User* table. Click of *Deposit* button invokes *Deposit* method of the corresponding *Account*.

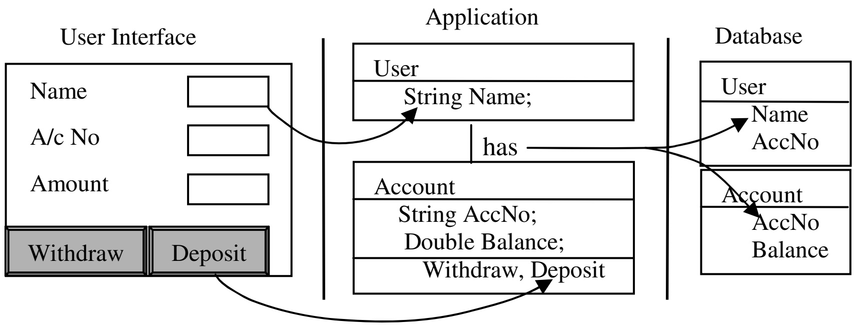


Fig. 7. Specifications for a banking system

The above example illustrates the advantages of using different notations to specify the different layers of an application. Making these specifications into instances of a single meta model allows us to specify the relationships between the different specifications. The notations proposed have well defined semantics. These properties allow specifications to be independently transformed into implementations that are guaranteed to integrate into a consistent whole.

6 Component-Based Development Process

A typical business application can be divided into a set of interacting functional units e.g. Foreign Exchange, Business Partner, Retail Banking of a banking system. A functional unit has high internal cohesion and interacts with other functional units in a well-defined manner. Typically, this functional decomposition is the basis for arriving at a development process with separate teams being assigned to separate functional units. Interactions between functional units manifest in a functional unit requiring some functionality provided by other functional units. One has to ensure consistency of such provider – supplier relationships during the development. Ensuring this consistency is a manual, effort intensive and error prone process.

We introduce *component* as an abstraction to model functional decomposition. A component has an *interface* and a *dependency*. A component interface is specified in terms of the model elements such as classes, operations, queries etc. A component explicitly models the components it depends upon. A component can only use the model elements specified in the interface of the component it depends upon. A component specification consists of model and code. The dependency relationship is honoured both in model and code. Explicit modeling of the dependencies enables automated consistency checking of provider – supplier relationship.

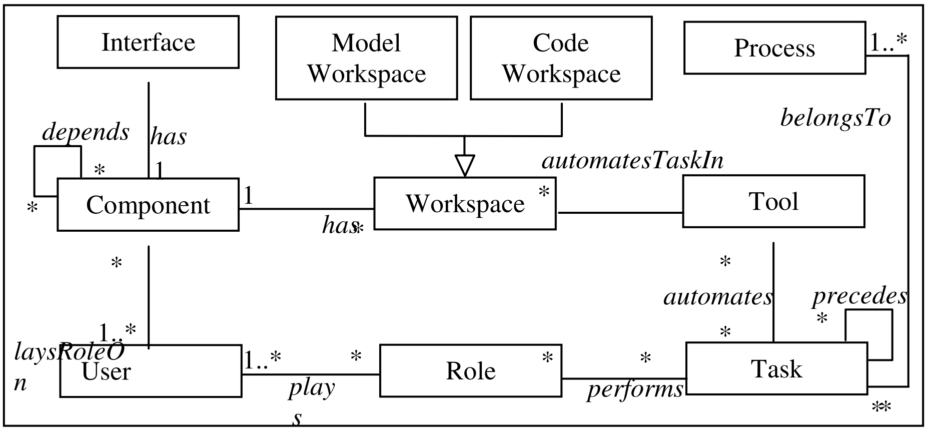


Fig. 8. Meta model for MDD framework

We introduce a component-based development process that supports the transition of a component through the various development phases namely analysis, design, construction, testing and deployment in a tool-assisted manner. The process provides a set of roles each responsible for performing a set of well-defined tasks on a component as shown in Fig. 8. A component has two associated workspaces namely model workspace and code workspace. Tools automate some of the tasks.

7 Discussion

The approach presented in this paper has been realized in our model driven development environment [1]. The approach has been used to develop several large business applications a representative set is summarized in the table below. The column *Domain model* refers to the domain classes and not to the implementation classes.

We discuss our experience in using this approach in these projects. Several projects had a product-family nature wherein a product-variant needed to be quickly put together and customized to meet the specific requirements of a customer. Model-driven development approach has helped in quickly retargeting the application functionality on multiple technology platforms. This was achieved using a relatively unskilled workforce as the technology and architecture concerns were largely taken care of by the tools. The tool-assisted component-based development process helped in early detection of errors that would otherwise have led to late-stage integration problems. Also, all the projects reported significant improvements in productivity and quality.

Project	Specifications			Generated code		Technology Platforms
	Domain model (no of classes / screens)	Size (kloc)	Kind	Size (kloc)	Kind	
Streight Through Processing system	334 / 0	183	Business logic, Business rules, Queries	3271	Application layer, Database layer, Architectural glue	IBM S/390, Sun Solaris, Win NT, C++, Java, ICS, MQ Series, WebSphere, DB2
Negotiated dealing system	303 / 0	46	Business logic, Queries	627	Application layer, Database layer, Architectural glue	IBM S/390, Win NT, C++, CICS, MQ Series, COM+, DB2
Distributor management system	250 / 213	380	Business logic, Business rules, Queries, GUI	2670	Application layer, Database layer, GUI layer, Architectural glue	HP-UX, Java, JSP, Weblogic, Oracle, EJB
Insurance system	105 / 0	357	Business logic, Business rules, Queries	2700	Application layer, Database layer, Architectural glue	IBM S/390, Sun Solaris, C++, Java, CICS, DB2, CORBA

The projects also reported a few problems with the approach. In model-driven development approach, part of the specification is in model form and part of it in code form. However debugging support was available only at the code level leading to difficulties in debugging. Also, the cycle-time required to effect a small change and verify its correctness was found to be significantly greater for the model-based approach then the traditional approach. However, the fact that a model-level change gets automatically reflected at multiple places in a consistent manner was appreciated.

Though we have illustrated the approach using a three-layer architecture, it lends itself to any architectural decomposition that has well-defined layers with well-defined relationships between them. The approach can be extended to support successive levels of refinement with guarantees of integrity at each level of refinement until a level is reached that can be automatically transformed into an implementation.

Many architectural and design strategies cut across the layers. In the present approach, this requires each tool to be aware of these cross cutting aspects. As a result, customizing for such cross cutting aspects requires consistent modifications to several tools leading to maintenance problems. We are investigating approaches to specify these aspects as modular building blocks that can be suitably composed.

8 Conclusions

We have presented the advantages of using different notations for different layers of application architecture. We have illustrated how having a single meta-model to describe the models corresponding to these notations and their relationships lends itself to an elegant implementation method. The implementation method allows independent transformations of specifications of the different layers and guarantees their integration into a consistent whole. The ability to develop an application by specifying and transforming each layer separately addresses the problem of scale. We have discussed our experience in using this approach in several large business applications.

References

[1] MasterCraft – Component-based Development Environment' Technical Documents, Tata Research Development and Design Center.

[2] **Sreenivas A, Venkatesh R and Joseph M,**. Meta-modelling for Formal Software Development in Proceedings of Computing: the Australian Theory Symposium (CATS 2001), Gold Coast, Australia, 2001. pp. 1-11.

Lessons Learned Applying UML in the Design of Mission Critical Software

Robert G. Pettit IV and Julie A. Street

The Aerospace Corporation,
15049 Conference Center Drive,
Chantilly, Virginia 20151 (USA)
[rob.pettit, julie.street]@aero.org

Abstract. This paper provides a series of lessons learned with respect to designing mission-critical software systems using the object-oriented paradigm and specifically with the application of the Unified Modeling Language (UML). The experiences captured in this paper are based on the authors' observations across multiple software systems and pertain to both the development processes and to UML modeling.

1 Introduction

High quality development of mission critical software systems is difficult. Many critical systems are developed, fielded, and used that do not satisfy their criticality requirements, sometimes with spectacular failures. Part of the difficulty of critical systems development is that correctness is often in conflict with (initial) cost.

This paper documents findings from a study to investigate best practices and lessons learned associated with mission critical software systems developed using object-oriented methods and the Unified Modeling Language (UML) [1-2]. As the industry standard for object-oriented modeling, UML is employed by nearly all modern object-oriented software projects. This paper specifically focuses on UML version 1.4 rather than version 2.0 simply because at this time, there is not a sufficient practitioner base in UML 2.0 to adequately capture lessons learned.

The organization of this paper is as follows:

- Section 2 establishes the context surrounding the lessons learned in this paper.
- Section 3 captures the lessons learned over the course of these development efforts in terms of both process and the application of UML models.
- Section 4 concludes the paper and discusses what can be done to further improve the state of the practice.

2 Current Status

The lessons learned documented in this paper are part of an ongoing effort to document experiences with applying object-oriented software design practices to real-time, concurrent, and embedded systems. These lessons learned are derived from several different software development efforts observed by the authors during the period of

2000-2004. The observed systems for this study were predominantly embedded software systems, often with real-time, reactive, and concurrent characteristics. Thus, the experiences captured in this paper tend to focus on those aspects.

While the anonymity of the observed projects must be maintained, we can generally categorize these projects in the domain of aerospace-related ground systems. The size and complexity of these projects was ranged from very small (less than 100 objects and 10 team members) to very large (over 1000 objects and over 100 team members). Regardless of size, each project represented some mission critical functionality of the ground system.

In terms of software design artifacts, all of the observed projects used UML 1.4 as the standard modeling language without applying any specialized profiles. Additionally, none of the observed projects used any special CASE tool for capturing real-time features or modeling embedded hardware components. Thus, as opposed to many existing works that focus on the shortcomings of UML and supporting CASE tools, the lessons learned in this paper only focus on how the basic features of UML can be better utilized in constructing mission-critical software designs.

3 Lessons Learned

This section documents the lessons learned in designing mission critical software systems with UML. These experiences are based on the authors' insight into projects dealing with large-scale software systems often in embedded environments and with real-time requirements and a high degree of concurrent processing. These lessons learned are captured in terms of two fundamental areas. Section 3.1 addresses experiences applied to the software development process and its effect on software design. Section 3.2 then presents lessons learned with the application of UML for modeling software designs within the targeted class of systems.

3.1 Process Lessons

In constructing object oriented software designs for critical systems, the software development process plays at least an equal role to that of the actual modeling activities. Decisions that are made from the very beginning of the software development effort can have a dramatic impact on the ultimate success or failure of that effort. The issues captured below represent the most common and important lessons learned with respect to the software development process as observed by the authors. The reader should note that the lessons in this section are not new revelations. Quite to the contrary, most of these points have been touted for many years. Surprisingly, though, most of these lessons learned are still not heeded in current software development efforts and are included to reiterate their importance.

3.1.1 Well-Defined Process

Simply employing a modeling language such as UML by itself is not sufficient. To successfully design a robust, maintainable, and flexible software system that meets the requirements and expectations of the customer, a well-defined process must be employed. The distinction between a well-defined development process and a general process framework also needs to be understood. Many projects opt for the latter, selecting such frameworks as the spiral development process [3] or the unified process [4]. While these frameworks are a good starting point, it is crucial for each project to capture the specific process flows, activities, and milestones that will be employed for their projects. This is nominally accomplished through the creation of a software development plan that documents not only the framework being applied, but also the specific process steps applied for the project.

3.1.2 Development Effort

Related to the process lessons learned, project managers must realize that simply adopting object-oriented practices does not reduce the development effort. In fact, for the first object-oriented effort, the development effort may actually increase due to the learning curve associated with adopting a new technology. The most positive experiences observed in this area are found with projects that have not necessarily changed their overall development effort, but rather have shifted more effort to up-front requirements definition and problem analysis. When these up-front activities were performed well, detailed design and implementation efforts were observed to be reduced at least marginally. Most notably, however, projects with a solid analysis model and software architecture were observed to reduce their maintenance efforts and reduce future efforts when adding features to their systems.

3.1.3 Improved Stakeholder Insight

One of the most immediate benefits observed from adopting a use-case driven UML design is the improved visibility to stakeholders. Through applying this highly visual modeling, software engineers are able to more readily communicate with systems engineers and even to the end customer. This results in an increased confidence with the system being developed and promotes a greater understanding of requirements early in the lifecycle. Furthermore, by using a standard language such as UML, engineers do not have to repeat the learning curve to understand the modeling constructs employed by each project.

3.1.4 Requirements Traceability

The lack of thorough requirements traceability is one of the most common and critical problem areas observed in current object-oriented development efforts. Often, requirements are traced to the use cases for a particular system or subsystem, but are not propagated to the individual design elements. When requirements are not completely traced to the specific design elements (e.g. classes, messages, statecharts, etc.), there is a tendency to lose focus as to the specific responsibility of the classes being designed. This can lead to costly changes late in the lifecycle and can also lead to incorrect or missing functionality in the delivered system. Additionally, gaps in requirements traceability complicate the testing and verification process, especially at the unit or white-box level.

3.1.5 Prototype Development

Many modern development efforts employ prototyping to achieve a better understanding for portions of the system under development. This is especially prevalent in the development of embedded software systems where prototypes are often used to gain increased understanding and confidence with respect to the embedded hardware with which the software must interact. While prototyping can provide valuable insight to the final system, extreme care should be exercised when applying the results of the prototype. Specifically, care should be taken to appropriately update the software design based on the results of the prototype. On projects where OO CASE tools were used, it has been noted that prototype code developed outside the tool actually hindered the development team's ability to use the tool properly [7]. It is the authors' observation that disconnects between design and implementation are one of the leading contributors to future maintenance and upgrade efforts within a software system.

3.2 Modeling Lessons

While the previous section focused on issues surrounding the software development process, this section focuses on the technical aspects of constructing object-oriented software designs for critical systems. Specifically, this section captures lessons learned with respect to modeling embedded, real-time, and concurrent software system designs using the UML. As noted previously, these lessons pertain to the basic features of UML 1.4 and do not depend on special extensions or CASE support.

3.2.1 Modeling Interfaces

Capturing interfaces to external devices is a critical element in the design of software systems dealing with embedded hardware. One simple, yet often overlooked approach is to develop a context diagram that clearly delineates the external devices with

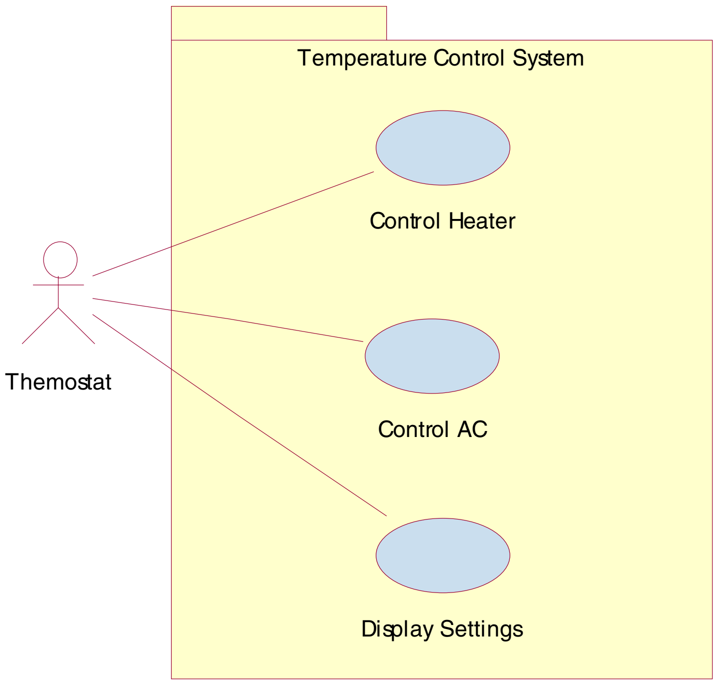


Fig. 1. Use Case Diagram

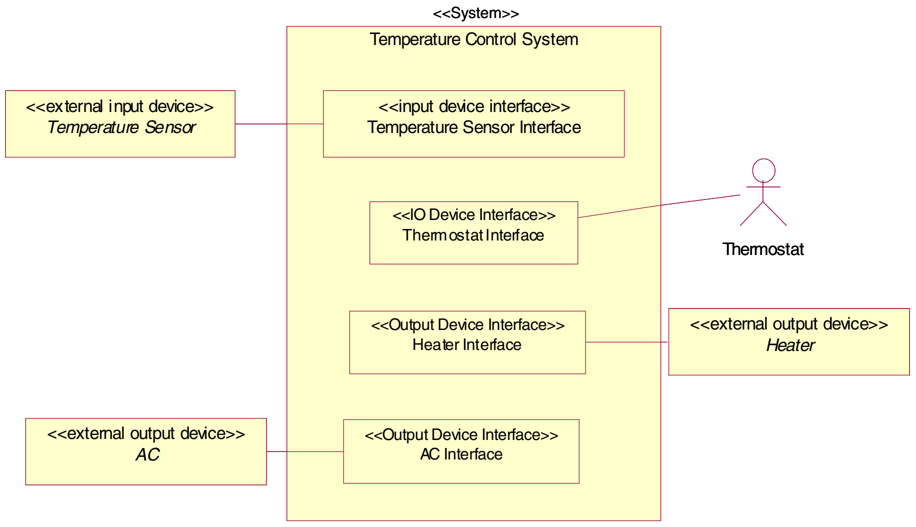

<<System>>
Temperature Control System
<<external input device>>
Temperature Sensor
<<input device interface>>
Temperature Sensor Interface
<<IO Device Interface>>
Thermostat Interface
Thermostat
<<Output Device Interface>>
Heater Interface
<<external output device>>
Heater
<<external output device>>
AC
<<Output Device Interface>>
AC Interface

Fig. 2. Context Diagram (NB: Some CASE tools force this to be drawn with the system as a package rather than a composite class as shown.)

respect to the software system. For each of these embedded devices, there should be a corresponding interface class whose sole responsibility is to encapsulate the specific interface characteristics of the device. By modeling each interface with its own class, the resulting software design improves flexibility and reduces dependencies on specific hardware interfaces. This allows for hardware to be replaced or upgraded while minimizing the impact to the software design.

In practice, many projects only identify system context through the use case model. This is often insufficient for embedded systems since not all embedded devices will be represented by an actor. Another common practice is to encapsulate hardware interface knowledge within a controller class. However, this practice reduces the flexibility of the software design by coupling the control logic with the specific interface details. Following the guidelines in the previous paragraph reduces this coupling and increases the flexibility and future maintainability of the software design.

To illustrate this point, consider a simple temperature control system where a thermostat is used to control a heater and an air conditioner (AC). The thermostat is responsible for maintaining the desired room temperature and for displaying the current settings. The use case diagram for this system is shown in Figure 1. While this use case diagram does capture the black box functionality of the system and identify the thermostat as an actor, it does not readily identify other embedded device controllers that the software must handle. If this diagram were augmented with a context diagram (drawn on a UML class diagram) as in Figure 2, we could then easily capture the fact that the system must interface with a temperature sensor, along with controllers for the heater and air conditioner. Note that this approach does not prevent software engineers from modeling detailed hardware information. Rather, it forces this detailed hardware information to be isolated in specific interface classes. Thus, performance and flexibility concerns can both be addressed.

3.2.2 Balancing Static and Dynamic Models

A common trend in object-oriented software design is to focus greater effort towards creating static models (class diagrams) than creating dynamic models (interaction diagrams). This practice results in an unbalanced design that, while providing a good data model, may not completely capture the behavioral aspects of events and messages that are prevalent in embedded software systems. Without adequately capturing this dynamic behavior, it is difficult to assess whether the final design will completely satisfy the functional or performance requirements of a system. Thus, when determining the development process to be used, care must be taken to allow sufficient attention to both dynamic and static aspects of the design as well as the ability to easily iterate between the models.

3.2.3 Choice of Interaction Diagrams

When constructing dynamic models, many projects choose to exclusively use sequence diagrams for capturing the message and event sequences of individual use case scenarios. While sequence diagrams provide a powerful mechanism to illustrate the sequence of events through a single scenario, they do not easily lend themselves to capturing the overall context of object interactions across multiple scenarios. For this view, the collaboration diagram actually provides greater clarity. By utilizing both forms of UML interaction diagrams, engineers can achieve a more complete description of both the sequence of events within a scenario and of the behavior across a set of scenarios.

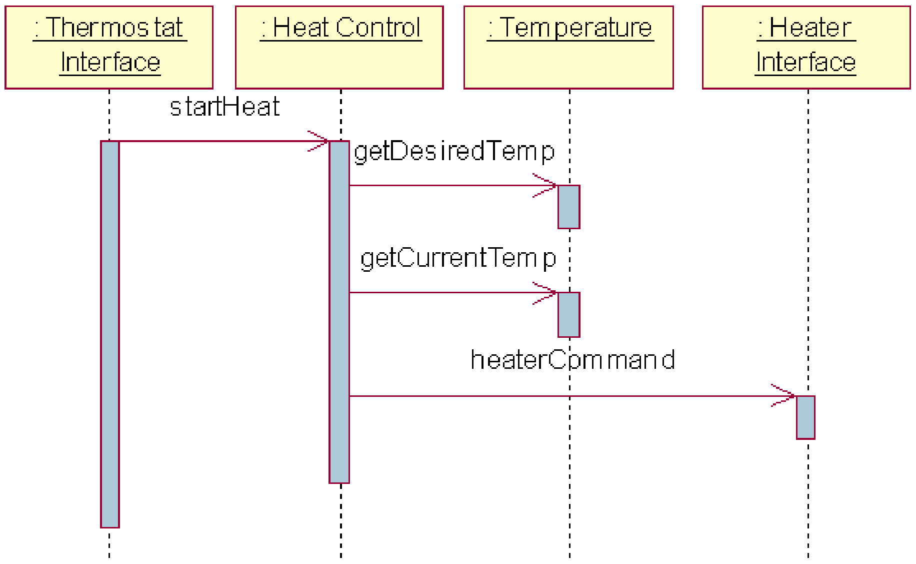


Fig. 3. Sequence Diagram

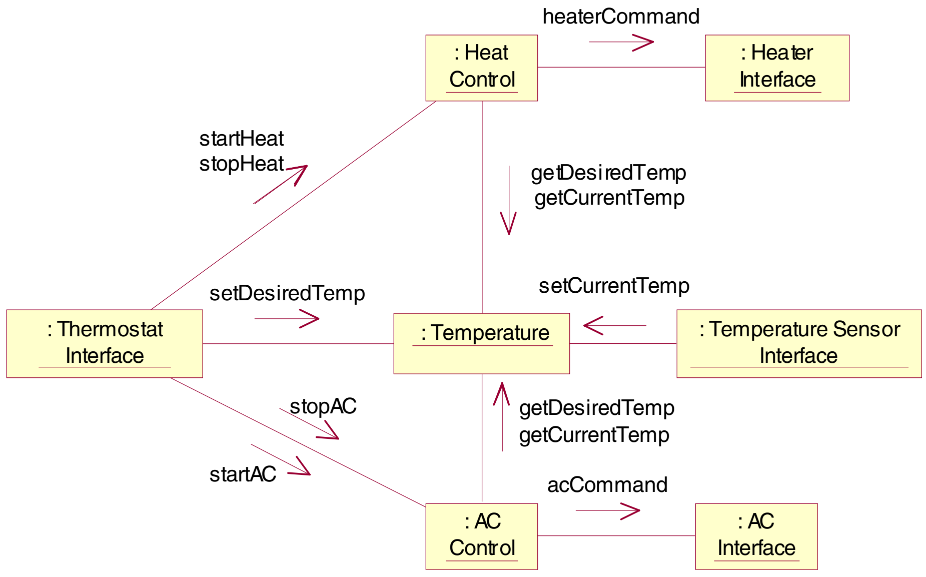


Fig. 4. Collaboration Diagram

As an example, again consider the temperature control system. In a typical design effort, each use case scenario would be realized in a sequence diagram such as that found in Figure 3 for the "control heat" scenario. While this sequence diagram certainly illustrates the sequence of events that occur for this scenario, it does not illustrate how the objects in this scenario might also be used in other scenarios. Instead of looking at individual sequence diagrams to derive this information, a collaboration diagram, as shown in Figure 4, can be used to illustrate the context of interactions across multiple scenarios or even for the entire software architecture (depending on size). As can be seen in Figure 4, this collaboration diagram shows the interactions that occurred in the previous sequence diagram and also adds information showing how the thermostat interface object interacts with the AC controller. It also shows how the temperature sensor is used to update the temperature object that is then used by both the AC and Heat controllers. Applying both of these dynamic views enhances the resulting design and further ensures completeness and consistency with respect to inter-object behavior.

3.2.4 Identification of Concurrency

As previously mentioned, the systems observed in this study were often composed of several concurrent tasks. However, many of these systems did not make use of the UML features to identify and handle concurrent objects. Rather than using the UML active object designation and the associated message communication constructs (e.g. asynchronous or synchronous) many projects chose to specify concurrent interactions and message behavior in separate (text) documents. One commonly cited reason for not relying on the UML active object construct is that active objects are somewhat limited in their ability to model detailed information like task management, priority, and protocols. However, the problem with not using the active object designation in design models is that this often leads to a disconnect between the as-built software and the UML design artifacts. A better solution would be to use the UML designation for active objects or at least use a stereotype to designate concurrently executing objects. Supplemental information can then be captured in outside documentation as necessary, but with this approach, the design accurately reflects the concurrent behavior of the objects.

Note that some projects have chosen to create custom stereotypes or UML extensions to handle these specifics [7-9], but these are non-standard and potentially not supported by CASE tools. UML 2.0 has additional profiles to address some of these limitations.

3.2.5 Statecharts

In the authors' experience, statecharts are one of the most underused UML diagrams in designing mission critical software system. The hierarchical statecharts employed by the UML offer significant expressive power for capturing the reactive, state-dependent behavior often found in these systems. Hierarchal states charts can be used to describe complex behavior in one class as opposed to having to decompose the static structure into several smaller classes, which can be confusing [7]. Statecharts should be constructed for each class that encapsulates state dependent behavior. The events and actions within the statechart should then be reconciled with the input and output messages for the class as captured in collaboration or sequence diagrams. From the projects we observed, those that judiciously employed statechart modeling experienced less ambiguity with the behavioral aspects of their designs and experienced a greater satisfaction of their software requirements.

3.2.6 Modeling Performance Requirements

Most mission critical software systems must conform to some set of performance requirements in addition to the functional requirements. However, performance requirements are often not modeled in the UML diagrams. For example, in the scenario captured by the sequence diagram in Fig. 3, it would be beneficial to show that there should be no more than a 1° Celsius deviation between current and desired temperatures. It would also be beneficial to show that there should be no more than a 100ms delay between detecting a low temperature event and engaging the heating unit. Ideally, these performance requirements would be captured using UML constraints. However, many case tools do not currently allow constraints to be added to interaction diagrams. Thus, designers often capture these requirements in other documents (if at all). At a minimum, though, these requirements should be annotated with a note on the interaction diagrams.

4 Conclusions

Object-oriented development practices offer significant benefits for designing modern software systems. In particular, UML offers many benefits to engineers with its multiple views and its status as the industry standard object-oriented modeling language. Unfortunately, many projects do not take full advantage of the features offered by UML. To address this problem, this paper documents some of the most common lessons learned in applying UML during the design of mission critical software systems, particularly those addressing the areas of embedded, real-time, and concurrent systems. These lessons learned are the result of ongoing efforts to document experiences gained applying object-oriented technologies within this class of systems. In capturing these lessons, it has been the authors' experience that process issues are as prevalent as modeling issues. By documenting and applying lessons learned through practical experiences, it is hoped that the quality of software designs will continue to improve.

References

1. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999
2. OMG. Unified Modeling Language Specification, Version 1.4, September 2001.
3. B. Boehm, "A Spiral Model of Software Development and Enhancement", *IEEE Computer* 21(5), May 1998.
4. I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. Reading, MA: Addison-Wesley, 1999.
5. R. Pettit, "Establishing Inspection Criteria for UML Models," *UML 2002 Tutorial Proceedings*, Dresden, Germany, October 2002.
6. H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Reading, MA: Addison-Wesley, 2000.
7. L.A.J Dohmen and L.J Somers. "Experiences and Lessons Learned Using UML-RT to Develop Embedded Printer Software." *PROFES 2002 Proceedings*, Rovaniemi, Finland, December 2002.
8. Moore, Alan. "Extending UML to Enable the Definition and Design of Real-Time Embedded Systems." STSC CrossTalk, June 2001.
9. Axelsson, Jakob. "Unified Modeling of Real-Time Control Systems and their Physical Environments Using UML", *IEEE Computer*, 2001.

System-on-Chip Verification Process Using UML

Qiang Zhu[1], Tsuneo Nakata[1], Masataka Mine[2], Kenichiro Kuroki[3], Yoichi Endo[3], and Takashi Hasegawa[3]

[1] Fujitsu Laboratories LTD., 1-1, Kamikodanaka 4-chome, Nakahara-ku, Kawasaki 211-8588, Japan
{shu.kyou, nakata.tsuneo}@jp.fujitsu.com
[2] Fujitsu Cadtech Limited, 2-3-9, Shin-Yokohama, KouHouKu-Ku, Yokohama, 222-0033, Japan
mine@fjct.fujitsu.com
[3] Fujitsu Limited, 1-1, Kamikodanaka 4-chome, Nakahara-ku, Kawasaki 211-8588, Japan
{kkuroki, y.endo, thasegaw}@jp.fujitsu.com

Abstract. In this paper, we propose a verification methodology for System-On-Chip (SoC) design using Unified Modeling Language (UML). We introduce UML as a formal model to analyze and formalize the specification. The specification and implementation validation can be performed systematically by introducing UML. We applied our method to a Mobile Media Processors SoC. We improved the quality of て the specification written in informal natural language through UML modeling techniques. The test scenarios and coverage metrics for implementation are derived from the UML model systematically. The result shows that our proposal is effective for eliminating errors from both specification and implementation.

1 Introduction

With the increasing complexity of hardware-software heterogeneous systems such as SoC, we have to face two crises in SoC design. The design crisis is caused by design productivity gap [1] due to chip complexity growing 58% but design productivity currently growing 21% annually. This means the current SoC design methodology cannot adapt to the growth of complexity of SoC.

With the increasing functionality and gate counts, we have to face another crisis for SoC design, named verification crisis. According to our experiences, more than 70% period of the SoC development lifecycle is used to verify the correctness of the design. Nevertheless, despite a huge effort of verification, most of chips must respin once or twice after they released. Unfortunately, each chip respin usually costs hundreds of thousands dollars. This indicates we need a new verification process for SoC development. Note that we use verification as the same meaning of test in software area in this paper because the notion of test always means diagnosis of circuits in the hardware community.

In the software community, the use case driven approach is proposed for requirements analysis. It is not only useful for clarifying the functionality of the system but is also is useful for system validation [2]. Object-Oriented Analysis and Design (OOAD) techniques [3] [4] can help designers to seamlessly integrate the result of analysis into the implementation. UML [5] is employed as a modeling language to characterize the results of analysis and design obviously, clearly and comprehensively. In SoC design, Hardware Description Language (HDL) such as VHDL [6], Verilog [7] are programming languages for implementing hardware at the Register Transfer Level (RTL) [8]. Recently, C/C++ based design flow was proposed using SystemC [9] or SpecC [10]. SystemC is a C++ library proposed to make an executable model from the algorithmic level to the RTL level by introducing timing, structure, and parallel description. SpecC is another modeling language for realizing an executable specification at the early design stage for SoC. SCE [11] is a development environment for SoC using SpecC. An object-oriented design process for system-on-chip using UML and SystemC is proposed in [12] through extending UML notation to represent parallelism, structure and timing. Such techniques tend to concentrate on the design process and make an effort to depart from RTL to the behavior levels so that designers can develop SoC more efficiently. However, these techniques have not provided good solutions for resolving the verification crisis.

For avoiding both design and verification crises, we propose a novel design and verification process from high-level specifications to RTL implementations using the Unified Modeling Language (UML), Component Wrapper Language (CWL) [13] and SystemC shown in Fig. 1. The key strategies of our approach are:

- Refining the design from requirements analysis to the RTL implementation incrementally.
- Integrating validation and verification processes not only for the RTL implementation, as well as for the specification based on formalized UML and CWL models.

The design crisis can be mitigated by introducing incremental refinement of design through system level functionality and performance analysis techniques [12] [14]. The verification crisis can be avoided through eliminating errors from the design specification at an early stage based on a formalized specification model.

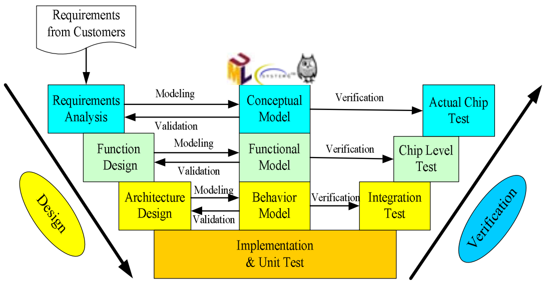


Fig. 1. SoC Design & Verification Process

In this paper, we focused on how to integrate UML and CWL into the verification process for a real SoC development in Fujitsu. We describe our verification strategies in Section 2. In Section 3, we introduce how to apply our process to Mobile Media Processors (MMPs) [15]. We show our application results in section 4 to discuss the effect of our approach. Finally, we give conclusions and future work of our activity.

2 Verification Strategy for SoC

An SoC contains functional hardware components or reusable Intellectual Properties (IPs), embedded CPUs, embedded memories, external interface controllers, software, mixed-signal blocks, and common buses. Its size should be more than 500K gates. Sometimes, a real-time OS is introduced in the CPU to develop more complicated software program. The internal RAM is usually used for storing large intermediate computation results for reducing the communication to outside of the system.

The verification strategy focuses on the structure of SoC. We emphasize three levels of verification:

(1) Components functionality
(2) Components communication
(3) Chip-level integration

Furthermore, at each level we use UML and CWL to formalize the specification of components, communication, and chip-level specification. The formal specification not only helps us remove the errors from the specification but also is useful to derive test scenarios, verification metrics systematically.

2.1 Modeling Functionality Using UML

For validating functionality of each component in an SoC, we focus on the specification as well as the implementation. In the specification, we confirm who are the users of the target component, what are the functions it performs, and how users use such functions. We employ UML to model such information captured from specification and validate that there are no incompleteness, inconsistency errors in the specification. Figure 2 shows the modeling flow that translates the specification into an UML model.

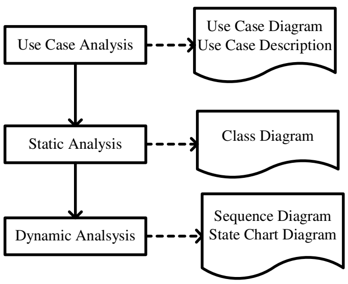


Fig. 2. UML modeling flow from specification

The UML modeling flow includes three steps:

Use Case Analysis captures users, functions, and their relationships using use case diagrams. Fig. 3 depicts an example of a use case diagram. Actors in use case diagrams describe the external components who want to utilize functions of the target component. Use cases express the functionality of target component. The stereotypes <<include>>, <<extend>> represent the relationship between use cases [16]. The boundary of the target component can also be modeled in use case diagrams using the system boundary notation. Use case diagrams can help us clarify who want to use the target component, what functions the target component provides and what are the relationship among actors-actors, use cases-use cases, and actors-use cases. Certainly, we can find the same information from traditional specification documents written in informal natural language. Unfortunately, those are usually dispersed in traditional specifications and not well organized. In some cases, the relationships among them are not clear due to ambiguities in the nature language specification. Use case diagrams can make them clear, compact and unambiguous.

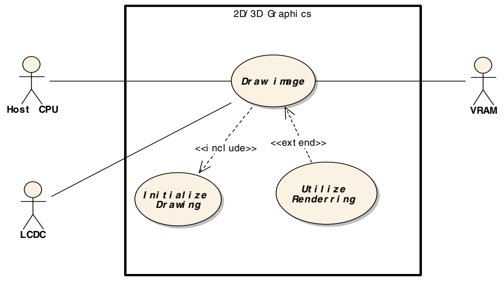


Fig. 3. An example of use case diagram

After clarifying use cases, actors and their relationships, we must know how actors use such functions. The usages of use case are represented by clarifying event flows among the target component and its associated actors. An event represents an input trigger from an actor to the target component or an output signal from the target component to an actor. A path of event flow expresses an interaction among the target component and actors while performing use case. The event flow for a use case is written in natural language, namely **use case description**. Table 1 shows an example of use case description.

In use case descriptions, we must clarify the following items for each use case: pre-condition, post-condition, basic path, alternative path, and exceptional path [17] respectively.

The use case description can help us clarify interactions among the target component and actors. Furthermore, pre-condition gives us restrictions before performing the use case, and post-condition provides the expected conditions after performing the use case. Actually, it is not feasible to find such information from traditional specification documents because designers always only consider the implementation of functionality without concerning themselves about third-party use. However, such design practice can cause errors due to incompleteness of the specification. In some cases, designers can carry over such errors into the implementation, which then requires enormous effort to remove in later stages of development.

Table 1. An example of use case descriptions

Use Case	Draw Image
Pre-condition	• Frame buffer must be cleared • Reset must switch off
Post-condition	• Image data output to VRAM
Basic path	1. This use case starts while Host CPU has already prepared the display list for drawing image. 2. Host CPU writes display list system's FIFO. 3. 2D/3D Graphics reads display list from FIFO and starts drawing image according to commands of display list. 4. 2D/3D Graphics outputs image data to VRAM.
Alternative path	At step 2 of the basic path, Host CPU writes display list that includes SYNC command to system's FIFO. At step 3 of the basic path, 2D/3D Graphics stops after reading SYNC command, and then 2D/3D graphics restarts drawing image after receiving a blank pulse from LCDC.
Exceptional path	At step 2 of the basic path, Host CPU writes display list includes undefined commands. At step 3 of basic path, 2D/3D graphics raises an error interrupt to Host CPU and clears system's FIFO.

Static Analysis concentrates on the structure and data types that appear in the specification. We use UML class diagrams to describe them and their relationships. We stipulate there are two types of classes in SoC specifications. One represents components, namely the control class. Another is the data type class that appears in input/output events. For example, we can find such classes from the use case description mentioned in Table 1. "Host CPU", "VRAM", "FIFO", "2D/3D graphics" should be objects of control classes and "display list", "image", "sync command" should be objects of data type classes. Fig. 4 shows an example of class diagrams. The class notated with the <<SoCModule>> stereotype indicates a control class and the <<SoCDataType>> stereotype represents a data type class.

Dynamic Analysis captures a system-level behavior by considering the input/output events in event flows for each use case. We use UML sequence diagrams and state chart diagrams to describe behaviors of the target component.

We use sequence diagrams to formalize event flows in use case descriptions written in natural language. The events in use case descriptions can be modeled with operations of a class. Fig. 5 shows an example of sequence diagrams for the alternative path described in Table 1.

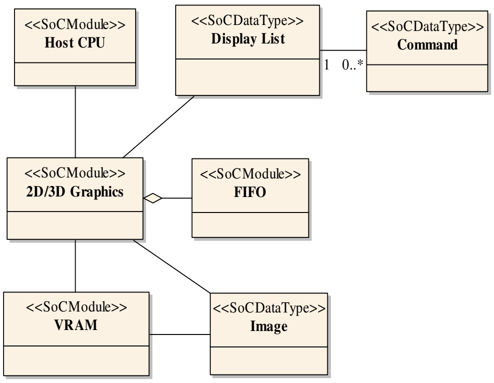


Fig. 4. An example of class diagrams

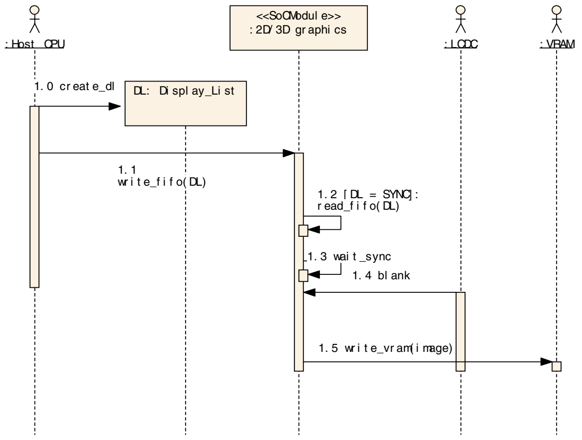


Fig. 5. An example of sequence diagrams

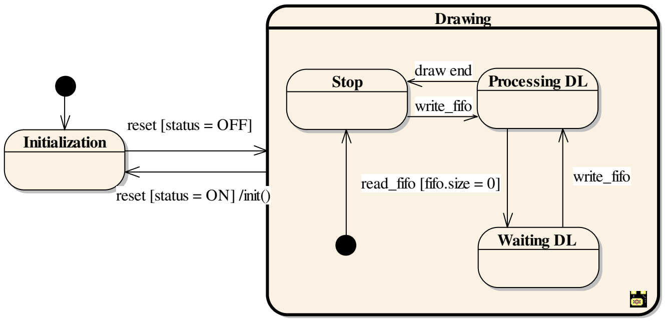


Fig. 6. An example of state chart diagrams

We employ state chart diagrams to clarify the state transitions at the system level for all event flows. From the analysis of all sequence diagrams, we can capture state transition from input/output events, and describe them with a state chart diagram. Fig. 6 shows an example of state chart diagrams. State chart diagrams can help us find the paths of event flow that are not listed in use case descriptions. This lets us validate whether the specification or UML model is complete or not.

2.2 Modeling Communication Using CWL

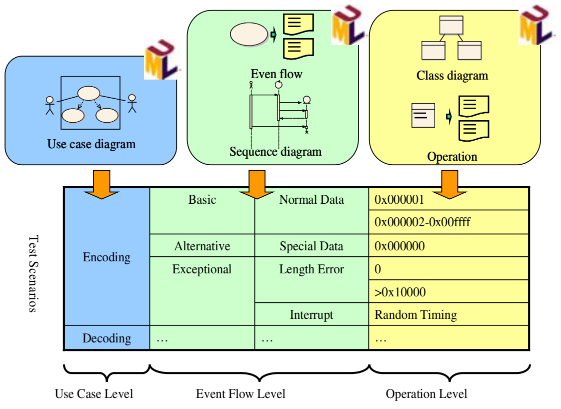
The analysis and modeling method mentioned above can be found in object-orient analysis and modeling techniques. We also adopt them into SoC developments. However, in SoC designs, communication among components is much different from software. We need a specification description for modeling the communication between the input and output ports of a component at the signal level. Unfortunately, we have not found a proper notation to model such properties efficiently in UML. Component Wrapper Language (CWL) [13] has been proposed to model the interface protocol of IP cores. We introduce CWL to model the communication protocol among components. The communication protocols are divided into the atomic transactions that are used from operations (events) modeled in sequence diagrams in the UML model as a method call. The CWL model translates such method call into the signal-level communications.

2.3 Validating Functionality of Components

We adopt two strategies for validating the functionality of components in our verification process:

- ♦ Validating the UML model to find errors in the specification due to incompleteness and inconsistency.
- ♦ Verifying the implementation using test scenarios and functional coverage metrics that are derived from the UML model.

In this paper, we focus on how to utilize the UML model to validate the implementation systematically. Fig. 7 shows the basic idea of our approach.



Encoding	Basic	Normal Data	0x000001
			0x000002-0x00ffff
	Alternative	Special Data	0x000000
	Exceptional	Length Error	0
			>0x10000
		Interrupt	Random Timing
Decoding	…	…	…

Fig. 7. Deriving test scenarios from UML model

First, we derive test scenarios from use case diagrams. The test scenarios at use case level must cover all use cases that appear in use case diagrams. Second, we focus on event flows and sequence diagrams to capture all paths of event flow for each use case. We list the test scenarios at the event flow level that include the basic, alternative, and exceptional paths. Finally, we extract parameters at the operation level from classes and their operations to determine the parameter values for a test scenario at the event flow level. Usually, the number of parameter values is so large that it results in a combinatorial explosion. In such cases, we use boundary value analysis [18] used in software testing that only choose maximum, minimum, typical values as well as boundary values from restrictions of data type classes. The functional coverage is a metric to measure how many test scenarios have been performed out of all the test scenarios derived from the UML model.

2.4 Validating Communication of Components

For validating the correctness of communications among components, we generate an HDL protocol checker from the CWL description using CWL2HDL tool [19] shown in Fig. 8. We insert the protocol checker among components to check whether there are any protocol violations of communications during functional validation of the implementation.

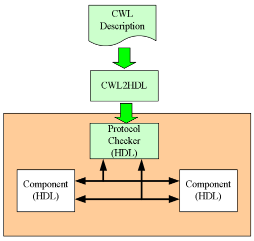


Fig. 8. Protocol checker for communication

We use transaction coverage, which is the state transition coverage of protocol checker to measure the coverage for communication protocols of the test scenarios.

After validation of components and their communications, we integrate them into a chip level verification. In chip integration testing, we also derive test scenarios and functional coverage from the UML model captured from a chip level specification to confirm the correctness of the specification as well as the implementation.

3 Applying for a Media Processing SoC

We applied our method to a Mobile Media Processors in Fujitsu [10]. In this section, we describe the details of our application.

3.1 Overview of the MMPs

Fig. 9 shows the architecture of MMPs. It includes MPEG-4 and JPEG hardware codec components, a two-dimensional tree-dimensional (2D/3D) graphics accelerator, and a camera interface (up to 2M pixels) with two YUV sensors, image scaling and rotation components, LDC interface and so on. To eliminate the need for external memory, MMPs incorporates 64Mbits of SDRMA as a system-in-package through a local memory controller. All functional components are connected with a local bus.

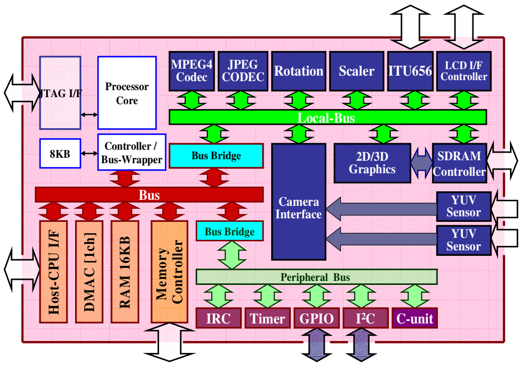


Fig. 9. The architecture of MMPs

3.2 Verification Process for MMPs

We integrated our process into a traditional design process for reinforcement of the verification process using UML and CWL. We organized a verification team that was separate from the design team. Designers developed the chip based on traditional design style: determining the chip level specifications, making the component specifications and their communication specifications, then implementing them into the HDL source code. Debugging the implementation and white box verification for each component was performed as usual. Note that all specifications created by designers are written in natural language. On the other hand, the verification team analyzed specification and translated them into UML and CWL respectively. While the verification team found errors due to incompleteness, inconsistency in the UML & CWL model, the verification team confirmed such errors with the designers to eliminate them from the specification or the UML & CWL model. After validation of the specification, the verification team derived test scenarios from the UML model to validate the implementation of components, communications, and chip integration with the black box verification.

3.3 Application Results

We took 6 months and 12 verification engineers to validate the specification and perform black box verification for each component, communication, and chip integration. We made code coverage, functional coverage, transaction coverage reach

100% for each component, their communication and chip integration. The chip released in about 1.5 months after finishing the verification and implementation. So far, we have not found any serious errors after the first chip released.

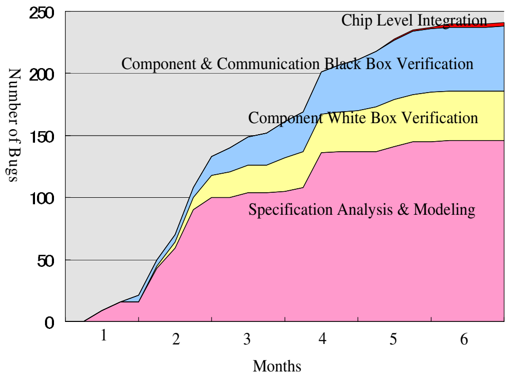


Fig. 10. The results for errors and their discovery time

Fig. 10 shows the progress for errors and their discovery time. The verification team found 132 errors due to incompleteness and inconsistency of specification in analysis and modeling phase. The design team found 40 bugs from the source code through white box verification. Then the verification team found 51 errors from the implementation with black box verification using test scenarios and test benches derived from the UML model and CWL protocol checker. In chip integration test, we only found three bugs from the implementation. These results indicate that most errors can be removed at an early stage of design, especially in analysis and modeling phase by our proposed verification process.

Table 2. Specification errors for components

Name	#Pages	#Error1	#Errors2	#EPP
Image Processing Component s	191	6	21	1.41
	21	6	4	4.76
	51	0	4	0.78
	28	2	3	1.79
	17	2	0	1.18
Control Component s	74	0	16	2.16
	80	4	2	0.75
	21	4	17	10.00
	44	1	11	2.73
	13	0	6	4.62
	22	0	9	4.09

Table 2 shows the number of errors for each component that we found in the specification during the analysis and modeling phase. The components named “Image Processing Components” indicate image-processing components include 2D/3D graphics, MPEG, JPEG codec etc. The components named “Control Components” are controllers such as “LCD Interface”, “Bridge”, “Host CPU Interface”, and so on. The “#Pages” shows the number of pages for each component specification written in natural language. The “#Errors1” shows the number of errors in the specification due to incompleteness. The “#Errors2” represents the number of errors due to inconsistency. “#EPP” shows the number of errors occurred for per 10 pages of the specification written in natural language. The results of Table 2 shows our process can effectively help us improve qualify of the specification before validation of the implementation.

Table 3 shows the results of bugs we found from implementation using test scenarios from the UML model. The “Name” shows the name of components as same as Table 2. The “#Scenarios” depicts the number of test scenarios for each component. The “#Bugs” shows the number of errors we found from the implementation. The number of scenarios in some control components is very high, because we used the random test pattern generation to assure that the combination of parameters were sufficient to test the all paths in these components.

Table 3. Implementation bugs for components

Name	#Scenarios	#Bugs
Image Processing Components	3,141	7
	629	3
	172	2
	1,069	2
	372	3
Control Components	995	9
	3,419	6
	168,556	3
	234	0
	> 50M	10
	57	5

4 Conclusions and Future Work

In this paper, we proposed a verification process that integrates UML and CWL to validate the functionality and communication protocols for SoC. We applied our method to a Mobile Media Processor developed in Fujitsu. We found 132 errors from the specification written in natural language by designers and discovered 51 bugs from the implementation with black box verification using test scenarios and coverage metrics that are derived from the UML model. The application results show our method is not only helpful to improve the quality of specification, but also useful to eliminate bugs from the implementation efficiently.

In future work, we will develop a tool, which could help us validate the completeness and consistency of the UML model automatically. Meanwhile, we will also improve our process through applying it to production SoC developments.

References

[1] Semiconductor Industry Association, *International Technology Roadmap for Semiconductors: 1999 edition.* Austin, Texas: International SEMATECH, 1999

[2] I. Jacobson, *Object-Oriented Software Engineering A Use Case Driven Approach*, Addison-Wesley Toppan, 1995.

[3] J. Rumaugh, M. Blaha, W. Lorensen, F. Eddy. *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[4] Baudoin, Claude & Hollowell, Glenn. *Realizing the Object-Oriented Lifecycle*. Upper Saddle River, NJ: Prentice Hall, 1996.

[5] OMG home page, http://www.omg.org/

[6] Standard VHDL Language Reference Manual, IEEE Std. 1076-1987, 1998.

[7] Thomas, Donald E, and Philip R. Moorby, *The Verilog Hardware Description Language*, second edition, Kluwer Academic Publishers, 1994.

[8] Standard for VHDL Register Transfer Level Synthesis, IEEE Std. 1076-6-1999, 1999.

[9] SystemC OSCI, http://www.systemc.org

[10] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, S. Zhao, *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[11] S. Abdi, J. Peng, H. Yu, D. Shin, A. Gerstlauer, R. Doemer, and D. Gajski, "System-on-Chip Environment: SCE Version 2.2.0 Beta Tutorial," TR 03-41, December 2003**.**

[12] Q. Zhu, A. Matsuda, S. Kuwamura, T. Nakata, and M. Shoji, "An object-oriented design process for System-on-Chip using UML", Proc. of the 15th Int. Symposium on System Synthesis (ISSS 2002), 1-4 October, Kyoto, Japan, pp. 249-254.

[13] Component Wrapper Language, http://www.labs.fujitsu.com/en/techinfo/cwl/index.htm

[14] P. Lieverse, T. Stefanov, P. van der Wolf, Ed Deprettere, "System Level Design with Spade: an M-JPEG Case Study," *IEEE/ACM International Conference on Computer Aided Design ICCAD2001*, pp26-32, November 2001.

[15] Press Release of Fujitsu Microelectronics America Inc., http://www.fma.fujitsu.com/newsArt.asp?code=033004b

[16] Object Management Group, *OMG Unified Modeling Language Specification 1.3*, 2001

[17] C. Larman, *Applying UML and Patterns: an introduction to object-oriented analysis and design,* Prentice Hall Jpan, 1998.

[18] E. Dustin, J. Rashka, J. Paul, *Automated Software Testing: Introduction, Management, and Performance*, Addison-Wesley, 2002.

[19] CWL2HDL, http://www.labs.fujitsu.com/en/techinfo/cwl/download_tools.htm

SoftContract: Model-Based Design of Error-Checking Code and Property Monitors

Luciano Lavagno[1], Marco Di Natale[2], Alberto Ferrari[3], and Paolo Giusto[4]

[1] Cadence Berkeley Labs, Berkeley, CA
[2] Scuola Superiore Sant'Anna, Pisa, IT
[3] PARADES, Roma, IT
[4] Cadence Automotive Team, San Jose, CA

Abstract. This paper discusses a *model-based design flow for requirements* in distributed embedded software development. Such requirements are specified using a language similar to Linear Temporal Logic which allows one to reason about time and sequencing. They consist of assertions which must hold for a design, given some assumptions on its environment. They can be checked both during simulation and, at least for a subset, even on the target. Of course the guarantee of correctness is ensured only as long as the assertion express the complete design intent, and the simulation stimuli cover all possible cases. While this is generally not true, the simulation-based approach is a practical manner to ensure correctness with a good degree of confidence, while avoiding the intricacies of software formal verification. Assertions related to deadline satisfaction can also be checked statically by a schedulability analysis tool. The key contribution of the paper is the extension to the *embedded software* domain of assertion-based verification, and the *automated generation of property-checking code* in multiple target languages, from simulation, to prototyping, to final production.

1 Introduction

Today, car manufacturers provide specifications to sub-system suppliers, who design software and hardware subsystems that may include mechanical parts (e.g. injectors and throttle bodies). In general, volumes are large, cost and dependability being major driving forces. Once the sub-systems are provided back to the car manufacturers, they have to be integrated on the car and then the overall system must be tested. If the car manufacturer detects errors during the extensive testing period, which includes driving under extreme conditions, a chain of engineering changes is initiated that may (and it often does!) cause major delays in the design. Such problems are traceable for the most part to software errors, because of incorrect understanding of the specifications and unpredictable side effects when the subsystems are interconnected. The loop is particularly painful since testing is done when the car is almost ready for its launch on the market.

This paper addresses directly this issue, and discusses a *model-based design flow for properties* in distributed embedded software design, thus extending the traditional accepted model-based design paradigm. The proposed methodology supports the definition of requirements on the performance and dependability of a real-time distributed system, as well as the validation that they are met in the fully implemented system. In this context, we first consider applications of automotive electronics that set stringent requirements in particular on dependability attributes such as safety, availability, maintainability, and also confidentiality, due to the complexity of its design chain.

Current model-based design flows, such as those based on Ascet-SD [1] or Simulink [2] specifications and Real-Time Workshop Embedded Coder [2] or TargetLink [3] implementations, emphasize automated transformations of specifications early in the design cycles, therefore reducing the risk of incorrect implementations. Yet they neglect automated transformations of properties. The basic tenet of the proposed novel flow is that both functional (e.g. relating I/O values) and non-functional (e.g. specifying performance requirements) properties, must be stated *formally* at the highest possible level in the flow, immediately deriving them from the informal requirements captured in a natural language. The traditional mechanism for representing functional and some non-functional properties, e.g. I/O rates, is the definition of a *testbench*, which verifies operationally that the properties are satisfied. This method is not efficient, because it is too implicit, non-declarative and partial.

Constraints that a design must satisfy are *decomposed, checked and propagated* along the design flow, whether it uses a top-down, bottom-up or V-cycle path including specification, implementation and integration. In particular, propagation entails *automated transformation* from one domain to another when crossing levels of abstraction (e.g. temporal logic formulae translated into simulation monitors and then into on-line error-checking software). Decomposition and checking, on the other hand, enable a clean *design by contract* between different parties involved in different design levels (e.g. system architect and software designer).

The goal of contract-based design is speeding up dramatically the design and improve the quality of embedded systems. The former is achieved by enabling a clear communication of requirements between various parties involved in the specification, design and validation of embedded systems. The latter is obtained by describing and automatically tracking satisfaction of constraints throughout the design flow, *including post-production and on-line* (run-time) *checking,* in a formal way.

1.1 Previous Work

Past work in this area, which traditionally belongs to the formal and semi-formal verification methods, can be identified both on the hardware and on the software side. On the hardware side, assertion-based verification is emerging as a promising evolutionary method to introduce formal techniques to specify and check properties starting from the Register Transfer Level, as opposed to merely checking equivalence between optimized and unoptimized designs or between layout and netlist. Recent standardization efforts, such as the PSL proposal by Accellera [4], aim at defining languages that are close to the way in which designers are used to model, e.g. Verilog and VHDL, and which provide a full range of options including full temporal logic, both in untimed (e.g. every request shall eventually be granted) and timed (e.g. every request shall be granted within 15 clock cycles) forms. The Rosetta work [5] also aimed at defining a very generic mechanism, based on sets and logics, to reason about properties of hardware designs.

On the software side, Hoare triples have been classically used to describe the pre-conditions that must hold in order for a statement to be executed correctly ("assumptions", in the terminology of this paper) and the post-conditions that are guaranteed to hold after the execution of the statement ("assertions", in the terminology of this paper). However, their use has been typically limited to imperative languages, and their full power in general required the availability of a theorem prover in order to check that the post-condition is indeed implied by the pre-condition and the statement logic. In this context, we are pragmatically more interested in defining properties that are useful within a specific domain, written in a user-friendly language, and easy to check by simulation or on a prototype, rather than being used to formally prove the correctness of a design. More recently, the Object Management Group has standardized the Object Constraint Language, which has similar goals, i.e. to precisely state requirements that objects, scenarios and software systems modeled in the Unified Modeling Language must satisfy. The OCL, however, is very expressive, and suffers from the lack of a standard executable semantics for the UML (which should be added in the upcoming UML 2.0 standard, also from the OMG). Thus it becomes suitable for automated checking and decomposition only if an application-dependent subset is chosen by a specific *UML profile*. For example, the proposed UML Profile for Schedulability, Performance and Time [6] (SPT) defines a subset of the OCL that can be used for representing deadlines, execution times, usage of shared resources and so on. While subsetting is not necessarily a disadvantage, since it improves expressiveness, still having to learn several sub-dialects of the same language for different tasks is more difficult than using, as in this proposal, a specially tailored one that is suitable for all the verification tasks in the chosen application area (real-time software implementation and verification).

Before the SPT standardization effort UML had been enriched with non-standard stereotypes and timing notations in order to provide ground for a-priori verification of timing constraints. Some examples are the MAST project [7] and the work by Saksena [8]. The latter originated from research on the ROOM methodology. It added a simple formalism for timing constraints to the standard port-based ROOM components. The proposed methodology and toolset allowed for the automatic generation of embedded SW and a-priori guarantees on the schedulability properties of components.

Another notable effort aimed at providing an integrated environment for expressing functional and non-functional constraints is the HRT-HOOD methodology for hierarchical object-oriented design [9]. HRT-HOOD components (objects) are characterized by timing attributes and constraints, which can be analyzed for schedulability at design time. In [10] Cornwell proposed the use of the Z formal language for expressing the functional behavior of HRT-HOOD components, thus allowing for the automatic generation of Ada95 code.

Finally, and most important, Real-Time Logic [11] is probably the best known formalism among real-time systems researchers for expressing timing constraints. In [12] Mok proved it amenable to early run-time checking of timing constraints. The SARTOR proposal for an integrated environment [13] makes use of RTL, together with AND/OR dataflow graphs and Modecharts for specifying the control, dataflow and concurrency domains of embedded applications. The integrated toolset aims at providing automatic generation of code and a priori timing analysis (guaranteed satisfaction) of timing constraints, but it lacks any kind of automated transformation from one domain to the other, and it does not.

In this work we use the Logic Of Constraints [14], which is a language reminiscent of various temporal logics (CTL, LTL and RTL) and which has been developed specifically to reason about various quantitative aspects of an embedded system (not just time). LOC is useful for our purposes, because it can be translated into simulation monitors for on-the-fly checking, rather than requiring full-fledged model checking, which suffers from inherent state explosion problems. Moreover, its semantics is based on sequences of events over signals, and it is thus easy to use for designers who are familiar with tools such as Simulink. This proved to be a key advantage with respect to more classical temporal logics such as CTL and LTL, which were designed more with protocol verification in mind. LOC moreover allows one to associate and reason about any annotation, not just time but also e.g. energy or memory, with events in the system.

1.2 Terminology and Conceptual Model

A *design* is a modeled piece of hardware and/or software, which must be implemented as a result of the design activities. A design can be represented as a *structure*, i.e. an interconnection of *components* (also called *modules* or *blocks*) connected via *nets* to each other's *ports* (mechanisms to communicate between blocks, such as shared variables or messages). Each component, and thus eventually the whole design, may have a *functional* model, describing how its output ports relate over time with its input ports. Both structure and functionality are described using any appropriate modeling language such as C, StateCharts, Simulink, Verilog, VHDL, and so on.

An *event* is an update of value (not necessarily a change of value, i.e. the updated value may be the same as the old one) of a port of a module of the design. For example, the arrival of a value from a sensor, the decision to change the state of a design component, or the generation of a command to an actuator are all events. Each event is annotated with a time of occurrence, and optionally with other quantities (such as energy) for which constraints can be specified. Although our definition of design is independent of the chosen Model-Of-Computation (MOC), for the sake of this paper we focus on the *Discrete Event* (DE) MOC for functional and performance modeling. DE is a particularly amenable to represent control and RT automotive applications, since it is a sort of a least common denominator between other MOCs which can be used to embed dataflow networks, Simulink networks, Hardware Description Languages, StateCharts and synchronous languages into a common semantic framework.

Events can only occur on explicitly defined *ports* of components (ports are the mechanisms through which blocks communicate), or on specifically exposed *viewports*. Viewports are internal aspects of the block, like *state* for example, that the designer chooses to expose about their internal behavior, which is otherwise hidden. This black-box semantics is essential for efficient implementation and decomposition, since prematurely exposing information about internal aspects of a design leads to poor portability, modifiability, re-usability, verifiability and optimizability. Black-boxing also improves security of a company's Intellectual Property, by hiding implementation details.

The *environment* of a design is a part of the whole system which cannot or need not be implemented by the considered team (e.g. the engine for the electronic control unit implementors, or the sensor sample conditioning filters for the control algorithm implementors). In other words, this paper considers a design flow in which the top-level model is (recursively) decomposed into sub-models, whose design must be carried out by different teams or individuals, possibly belonging to different companies. Unambiguous communication between these teams or individuals, by means of assertions on the design that they must guarantee by implementation and assumptions on the environment that they can make, is one of the key advantages of this proposal over the state of the art.

A *property* is LOC formula, involving events and their annotations (e.g. time of occurrence), which must be true, and which can play different roles depending on the context. An *assertion* is a property which must be guaranteed to hold by a design. For example, the statement that the latency between an input and an output event must be less than 0.1 msec is an assertion. An *assumption* is a property which limits the set of environment behaviors to be considered, and thus exhibits some freedom that can be exploited by knowing that some cases can never occur. For example the statement that the maximum rate of arrival of input events is 1 per msec is an assumption.

Quite often, a *requirement* on a design component is expressed as a pair including: an assertion that is assumed by users of the component to hold, and guaranteed by its implementer to hold, and an assumption that is assumed by the implementer of the component to hold, and must be guaranteed by its users to hold, as illustrated by the following simple example. First of all, the designer in charge of assigning priorities to tasks running on a real-time executive can make *assumptions* on the maximum rate of arrival of events triggering them and on their WCET, and must satisfy *assertions* on the priority ranking (e.g. based on Rate Monotonic Analysis). Then the team who is in charge of implementing the tasks can make assumptions on the maximum rate of arrival of events and on priorities, and must satisfy assertions on their WCET. Finally, the integrator of the control unit in the car can assume priorities and WCETs and must satisfy assertions on event arrival rates.

A *monitor* finally is a component of a design whose main task is to verify that an assumption or an assertion on another component or set of components is satisfied. Monitors are executable checkers that can be used in simulation, prototyping and production code in order to ensure that the design contracts are respected. A key aspect of the proposed design methodology is the ability to derive various kinds of monitors for the various stages (simulation, formal verification, prototyping, production) from a single specification (model-based contract design). This ensures a consistent flow of information between various phases of design, verification and usage of components of an embedded system.

2 Design Flow

In the proposed design flow, the requirements on a design are first specified as assertions which must hold, given some assumptions on its environment. In order to be able to define such assertions and assumptions, one must have defined a skeletal structure for the design, at the very least the I/O ports with which it communicates with its environment. Assertions are *checkable* only when the functionality of the design has been specified. Some of them, e.g. those related with timing, are checkable only when the functional model has been annotated with performance information, so that the time information attached to events reflects the effects of the underlying *architecture*.

These requirements can be used both bottom-up and top-down. Bottom-up, they clearly specify the contract that the implementer promises to obey with respect to the users of a component. Assertions are guaranteed provided that assumptions are satisfied (e.g., this piece of software written in C computes the response with a precision of 1% provided that "int" variables have at least 32 bits). Top-down they specify requirements that the implementer must obey, and state the assumptions he can make on the users of the component.

An essential aspect of a *bottom-up* design flow is the *composition* of assertions on individual components, while checking that the used components guarantee each other's assumptions. A full-fledged compositional proof methodology would require theorem proving, an expensive proposition today even for safety-critical applications. More practically, monitors can be used to trace the requirements throughout the lifetime of a component. This is already common practice for safety-critical embedded software, e.g. in the automotive industry. In this case, code devoted to verifying that the input values received by a piece of code match the assumptions made by the designer of that piece of code, and that only legal values are produced as a result of the internal computations, can constitute a very significant portion of the total software content of a design. For example, governments have imposed regulations for the automotive industry that limit the level of chemical emissions from car engine exhausts. In order to comply with these regulations, a vehicle must satisfy the European On Board Diagnostics, a standard which imposes a set of properties of the system that must hold and are checked at run-time. This is implemented through a set of monitors allocated to the different electronic control units, which check relevant values of the state of the software (variables). These monitors are typically coupled with other components that implement recovery and logging in case of violations.

One of the innovations of our flow is, as discussed above, the use of the very same description of an assertion or an assumption (quite often they come in pairs, describing the conditions under which a given property is guaranteed to hold) for the various phases of the design. This is essential in order to ensure precise contractual obligations between parties in the system design flow. It also dramatically eases handoff points between teams or companies in the design flow, by making requirements explicit and formal, and speeds up implementation of the final code, by automatically generating the required monitors in the given context, from simulation to run-time.

In *top-down* design, on the other hand, requirements on the global I/O of the system are decomposed into sub-properties that must hold for each component of the design. The collection of sub-properties on other components, not under design by a specific team, together with assumptions on the global top-level environment, become the set of assumptions that an implementer can make on his component's environment, as illustrated in Section 3.

2.1 Property Specification Language

Logic Of Constraints [5] is a formalism designed to reason about execution traces. It consists of basic relational, Boolean and implication operators, with additions that make it possible to specify system level quantitative functional and performance constraints without compromising the ease of analysis. The basic components of an LOC formula are: events (defined above), the index variable i and annotations:

1. Annotation: each event may be associated with one or more annotations. Annotations can be used to denote the time, power, area, or any value related to the event. E.g., Display$[i-5].t$ denotes the t annotation (by convention time, while annotation v represents its value) of the $i-5$-th event of the Display port.
2. Index variable: LOC permits only one event index variable i, a positive integer, in a given expression (the limitation helps ensuring checkability in bounded memory). Index expressions of events may be any arithmetic operations involving i and constants, e.g. Display$[i-5]$, Stimuli$[i]$.

LOC can be used to specify some very common and useful real-time performance constraints:

– rate: E.g. "Displays are produced every 10 time units":
Display$[i].t$ – Display$[i-1].t == 10$
– latency: E.g. "Display is generated no more than 25 time units after Stimuli":
Display$[i].t$ – Stimuli$[i].t <= 25$
– jitter: E.g. "every Display is no more than 4 time units away from the corresponding tick of the real-time clock with period 10":

$$\text{Display}[i-1].t - (i) * 10 <= 4$$

– throughput: E.g. "at least 100 Display events will be produced in any period of 1001 time units":

$$\text{Display}[i].t - \text{Display}[i-100].t <= 1001$$

– burstiness: E.g. "no more than 1000 Display events will arrive in any period of 9999 time units":

$$\text{Display}[i].t - \text{Display}[i-1000].t > 9999$$

– maximum rate of change: E.g. "the (discrete) derivative of the value of S shall not exceed 10":

$$(\text{S}[i].v - \text{S}[i-1].v)/(\text{S}[i].t - \text{S}[i-1].t) < 10$$

For a LOC formula to be *formally proven* for a design, it needs to hold for all possible traces and all values of the index i, as it appears in the index expressions of the formula. For a formula to be *checked* for a particular simulation trace, it needs to hold for that trace only and all values of i. In the rest of the paper we are concerned only about checkability.

Both assertions and assumptions are expressed as LOC formulae to be checked. Their respective violation, however, is the sign of a breach of the contract by different parties (roughly speaking, if an assertion formula is not checked, i.e. it is violated, then it is my fault, while if an assumption is violated, then it is somebody else's fault). LOC formulae are, by construction, easy to check during simulation. It is also possible to generate code that checks them at runtime on a prototype. It may even be possible to check them at runtime on the real system, if their satisfaction is vital to the correct operation of the system. Logging these data on the target system is very useful in order to enable maintenance personnel to determine the state of the car and of its components, and to decide whether some intervention is required ("design for serviceability"). In addition, violations of assumptions or assertions can be used at runtime to trigger driver notifications and to enable default "safe" behaviors of the embedded controllers.

2.2 Target Language Translations

The properties specified using the language above can be translated automatically, into:

– Off-line database query code, which checks that both assertions and assumptions are satisfied on a given set of simulation traces. Probes are automatically generated and instantiated in order to collect enough information to answer the queries corresponding to all the properties being checked. The example described in Section 3 was checked in this off-line mode of design by contract.
– On-line monitor modules written in whatever simulation language is used for design verification. These monitors emit error messages when the assertion or assumption is violated, as well as a warning at the end of the simulation if an assertion or assumption is neither satisfied nor violated.
– On-line code to be integrated within the software tasks, to which the ports referenced in the property text belong. Integration of the code into specific "supervisory tasks", running under RTOS control and having access to local variables of other tasks, can also be generated in a second phase, for properties that refer to ports of components mapped to different tasks. This code can be used in a prototype, for debugging in the field, at least for that portion of property-generated code that is not intrusive and does not cause excessive load for the target CPU. Note that taking a consistent snapshot of the state of a distributed system may be very expensive, or even just impossible. All properties selected for runtime checking on the target must thus be *local*, with respect to the mapping onto the chosen architecture.
– Off-line and on-line hardware-assisted property checkers, using in-circuit debuggers or on-chip real-time tracers. The hardware resources provided by the these devices strongly limit the number and complexity of the properties that can be concurrently checked.

3 A Design Example

For the sake of illustrating our proposal, we describe an example of a safety critical application, typically implemented on a distributed multi-cluster ECU architecture. The application is a simplified version of an Adaptive Cruise Control (ACC), shown in Figure 1. The ACC includes "regular" cruise control features, but must also automatically decrease the speed of the vehicle, if an obstacle is detected at a distance less than the safety distance threshold. In this case, actuation signals are automatically sent to the brake system and to the engine control system.

The functional model that we used includes models of the driver, the radar system, the engine, and the brake. The ACC algorithm determines the gas pedal position (therefore replacing the driver) based upon the vehicle speed, the distance between the vehicles, and their relative speed. The control strategy is defined by the ACC Finite State Machine.

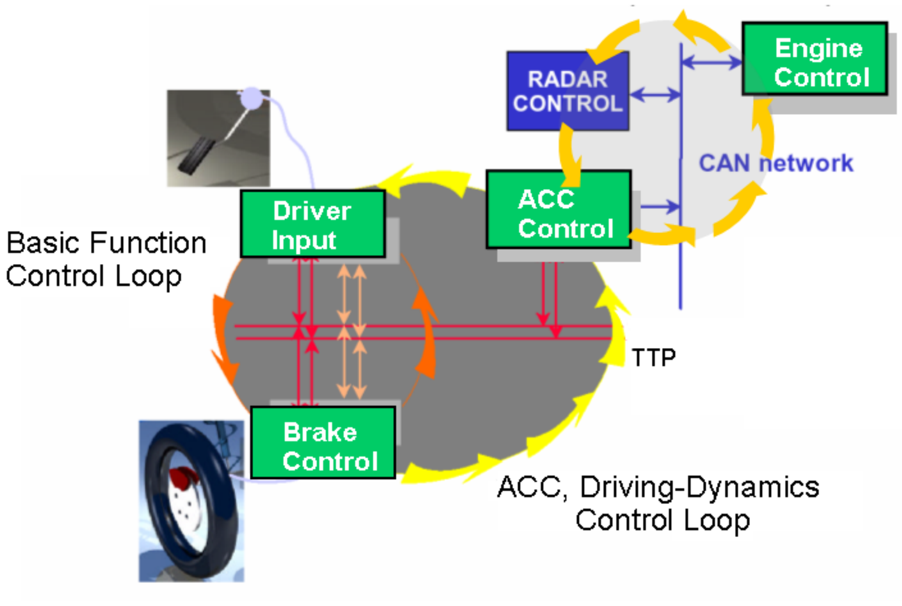


Fig. 1. The adaptive cruise control application

Based on choices from the driver, it decides which position of the gas pedal is provided to the engine control. The position may be determined either automatically, if the FSM is in the state “ACCon”, or by the driver.

3.1 Some Simple Properties

An important safety feature of our algorithm, that can be used to test the contract-based design flow, is that the current value of the gas pedal is retained in case the new position determined automatically is very different (for example due to data corruption) from the current one. This is expressed by the following LOC property:

```
define limit_change (comp, act, thr) {
  abs (comp[i].v - act[i-1].v) > thr ->
  act[i].v == act[i-1].v }
```

instantiated as the following requirement (assumption plus assertion):

```
assume FSM.State[i].v == ACCon
assert limit_change (FSM.GasPedalPositionFSM,
  FSM.GasPedalPositionACC, FSM.threshold);
```

Here `FSM` is the name of the block whose inputs and outputs are used in the property, `state` is a viewport exposing its state, `GasPedalPositionACC` is the output of the automated cruise control block (and input to the `FSM` block) which determines the required position to decelerate smoothly when required, `GasPedalPositionFSM` is the output of the `FSM` block which goes directly to the actuator, and `threshold` is a parameter which must be tuned on the prototype car in order to provide a smooth and safe driving experience. Finally, `->` denotes logical implication.

Another assertion that was checked in this design, using the LOC database monitors, is the following: if the distance between vehicles goes below a given threshold, then within 30 seconds the distance will be again above threshold.

```
define rate (g, thr, tol) {
  abs (g[i].t - g[i-1].t) < thr + thr * tol and
  abs (g[i].t - g[i-1].t) > thr - thr * tol) }
define slowdown (dist, thr, delta) {
  dist[i-delta].v < thr -> dist[i].v >= thr; }

assume ACCCore.Speed - Radar.OtherVehSpeed < 10
  and rate (ACCCore.speed, 0.001, 0.01)
assert slowdown (ACCCore.distance,
  ACCCore.threshold, 30 / 0.001);
```

Here we assume that the difference between vehicle speeds is less that 10m/s, otherwise, the only safe option for the driver is to brake by himself (this is *not* a drive-by-wire system, only an enhanced cruise control). Here `ACCCore.speed` is the speed of the current vehicle (an input to the ACC controller `ACCCore`), `Radar.OtherVehSpeed` is the speed of the other vehicle, as measured by the `Radar`, `distance` is their distance and `threshold` is a parameter defining the distance at which the speed must begin to be reduced. Time is measured here in terms of discrete controller invocation intervals, which is consistent e.g. with the Simulink semantics, and assumptions on the rate establish the relationship between invocations and time. For example, since the ACCCore model is invoked once every millisecond and the tolerance `tol` on the invocation rate is 1%, the index difference 30 / 0.001 refers to a time interval of 30 seconds plus or minus 1%.

Debounce assertions are important to correctly evaluate Boolean signals produced by the environment. When a switch is pressed, the output signal oscillates until it reaches a new stable value. The debouncing functionality guarantees that only the final value of the switch signal is used as input value. In our example, the switches that turn on and off the cruise control and the adaptive feature must be debounced before evaluation. The requirements to debounce a switch in a time window of 200ms can be expressed as follows:

```
event EdgeSwitch { Switch[j-1].v!=Switch[j].v }
assert EdgeSwitch[i+1].t-EdgeSwitch[i].t > 0.2;
```

This example uses an “event definition” facility of LOC, which allows one to define new events based on the occurrence of logic and relational conditions on existing events.

3.2 Assertion/Assumption Decomposition

We will now consider an example of how decomposition of assertions into pairs of assumptions and assertions can be used to define and verify the interface between two teams or companies working on two portions of the system. The adaptive cruise control must guarantee a certain degree of comfort during cruise. For instance the vehicle should not accelerate or decelerate, after reaching the cruising speed, by more than a 0.5 m/s^2, which can be expressed with the following assertion:

```
assert FSM.State[i].v == ACCon =>
  abs(Acceleration[i].v) < 0.5;
```

The overall system, as shown in Figure 1, is decomposed into ACC, Engine control and Brake control. The ACC provides the gas pedal position to the Engine control, which translates it to a request for a given amount of torque. The Engine finally produces the torque. The previous assertion, checked at run-time, would inform the designer if a violation on the vehicle acceleration occurred, but would not explain if this was due to a design error of the engine control or of the ACC control. If the two control units are built by different sub-system makers, it would be problematic to pinpoint the cause of the error in the design.

Following our methodology, the assertion should be decomposed into three parts:

1. an assertion on the torque requested by the ACC,
2. an assertion on the torque provided by the Engine control and the engine, and
3. an assertion on the relation between vehicle acceleration and torque.

The third assertion is always satisfied in a given gear, since it checks the inputs and outputs of a mechanical system, that is the powertrain of the vehicle. In this case, a torque smaller than 20 ensures an acceleration smaller than 0.5. The first assertion on the behavior of the ACC can thus be expressed as follows:

```
assert FSM.State[i].v == ACCon =>
  abs(ACC.TorqueRequest[i].v) < 20;
```

The Engine control unit maker is using the same property as an assumption, instead of an assertion, checking that the torque request, when the cruise control is on, is limited as specified and agreed. The second assertion thus is expressed as follows:

```
assume FSM.State[i].v == ACCon
  and abs(ACC.TorqueRequest[i].v) < 20
assert abs(Engine.Torque[i].v) < 20;
```

A violation of the vehicle acceleration is now shown by different checkers, and the sub-system causing the violation is easily found, even before system integration.

More complex comfort assertions can be efficiently added to the design, such as checking the jerk (i.e. the rate of change of the acceleration) of the vehicle, hence the rate of change of the generated torque.

The design described here was created using the Cadence Automotive System Design Platform (also known as SysDesign). Plant models were imported from Simulink via a special Real-Time Workshop target. The Engine control model along with the task structure was imported from Ascet-SD [1], a model based design environment for algorithmic development, with code generation capabilities for both prototype and target. The definition of the target multi-ECU architecture, the task allocation to the ECUs, the bus modeling and the simulation were performed in SysDesign.

Properties were checked automatically using a tool which compiles the LOC formula into a fragment of C code which reads the SysDesign simulation database and checks the validity of the formula off-line over a simulation run.

3.3 Lessons Learned

Although this project is currently at the research stage, and it has not been applied to any real-life example, its motivations stem from the observation of the current state of the art of model-based design in the automotive world, and of worldwide trends in the car electronics industry.

The key observation that we made during both the experiment described in this section, and during previous attempts at defining a model-based design flow for properties, is that the *language* used is extremely important, i.e. it is not just syntactic sugar. While this work is the latest (and certainly not the last) one in a long stream of property-based formal modeling approaches, we believe that it is unique in that the language used fits very well the working habits of designers that are supposed to use it. The notions of events, sequences and indices are familiar to everyone involved in discrete control, implemented on a computer. Hence the Logic Of Constraints is easy to use, much easier than forms of temporal logics or higher-order logics. It is, however, powerful enough to express properties of interest for a significant example, and we could easily generate efficient code for checking it both on-line and off-line.

In the future, practical application of the methodology and language proposed in this paper will require the definition and implementation of modeling mechanisms that are even more user-friendly, e.g. taking the form of a Simulink block-set. It will also require cooperation with companies and groups that are expert in the area of data logging and monitoring, and of parameter tuning, since they have the capability to extract the atoms on which the LOC is built, and hence are essential in order to efficiently check properties both on-line and off-line on the target system, both in the prototyping and in the production phases.

4 Conclusions

This paper proposes a *model-based design flow for assertions and assumptions* that together ensure the correctness, both functional and non-functional, of a complex embedded system. The paper uses examples, terminology and scenarios from the automotive software domain, but the flow is applicable to any safety-critical mixed hardware/software system. Assertion-based verification is becoming a cornerstone of hardware design. What is new in the case of safety-critical embedded systems is the extension to the *software* domain of assertion-based verification, and the *automated generation of code* in multiple target languages, from simulation database queries, to simulation monitors, to prototyping, to final production. This leads to:

- faster time-to-market, by reducing design iterations,
- real contract-based design between specifiers (system architects), implementors (software designers) and integrators, by allowing
  - fast verification by the sub-system providers that the assertions made by the architect on sub-systems are satisfied and
  - delivery of partial assumptions and assertions from sub-system providers to system integrators for earlier verification of end-to-end assertions.

– faster implementation, thanks to automated target code generation for assumption and assertion checking,
– safer implementation, due to the formal property specification mechanism.

In the future we are planning to explore the use of assertions and assumptions for automated testbench generation, e.g. by constraint solving.

References

1. Ascet-SD, E.: (2004) http://www.etas.de.
2. Simulink, T.M., StateFlow: (2004) http://www.mathworks.com.
3. dSPACE TargetLink: (2004) http://www.dspace.de/.
4. Language, A.P.S.: (2004) http://www.accellera.org/.
5. Alexander, P., Kong, C., Barton, D.: Rosetta usage guide (2003) http://www.sldl.org.
6. Group, O.M., ed.: UML Profile for Schedulability, Performance, and Time. OMG document ptc/02-03-02 (2002)
7. Medina, J., Harbour, M.G., Drake, J.: Mast real-time view: A graphic uml tool for modeling object-oriented real-time systems. In: Proceedings of IEEE Real-Time Systems Symposium. (2001)
8. Saksena, M., Freedman, P., Rodziewic, P.: Automated implementation of executable object oriented models for real-time embedded control systems. In: Proceedings of IEEE Real-Time Systems Symposium. (1997)
9. Burns, A., Welling, A.J.: HRT-HOOD: A design method for hard real-time. Journal of Real-Time Systems **6** (1994) 73–114
10. Cornwell, P.D.: Reusable Component Engineering For Hard Real-Time Systems. PhD thesis, University of York (1998) YCST-98-04.
11. Jahanian, F., Mok, A.: Modechart: a specification language for real-time systems. IEEE Transactions on Software Engineering **20** (1994) 933–947
12. Mok, A., Liu, G.: Early detection of timing violation at runtime. In: Proceedings of IEEE Real-Time Systems Symposium. (1997)
13. Puchol, C., Mok, A.: Integrated design tools for hard real-time systems. In: Proceedings of IEEE Real-Time Systems Symposium. (1998)
14. Chen, X., Hsieh, H., Balarin, F., Watanabe, Y.: Automatic generation of simulation monitors from quantitative constr aint formula. In: Proceedings of Design Automation and Test in Europe. (2003)

Tailoring IEEE 1471 for MDE Support

Eric Jouenne and Véronique Normand

THALES Research & Technology,
Domaine de Corbeville 91404 Orsay, France
{eric.jouenne,veronique.normand}@thalesgroup.com

Abstract. THALES is a supplier of complex systems in the Defence and Aerospace domains. A number of THALES Units are currently involved in transitioning parts of their systems and software engineering processes from document-driven to model-driven engineering (MDE). MDE puts models on the critical path of system and software development, turning the role of model from contemplative to productive. Putting MDE into practice in operational contexts intrinsically puts heavy demands on the tooled-up process.

A first challenge is to support MDE tooled-up process definition, implementation, assembly, and deployment all the while addressing industrial concerns for adaptability, maintainability and scalability. A second challenge is to be able to introduce MDE innovations in existing development processes at mastered cost, all the while managing the legacy.

The MIRROR Pilot Program was launched three years ago in THALES to address these challenges. The concept of MDE methodological component was identified as a building block for supporting the definition and building of MDE tooled-up processes. An implementation approach for this concept is developed here within the context of a THALES industrial case, based on an extension of the IEEE Recommended practice for architectural description of software-intensive systems, IEEE Std 1471-2000.

1 Introduction

The THALES business core is in supplying complex systems, mainly in the Defence and Aerospace domains. These systems have long life cycles; their development often requires large multi-disciplinary teams (system, software and/or hardware engineering). System and software engineering processes are the backbone of THALES development. It is through definition, formalization, and development process respect that system development maturity is achieved [4].

The THALES development projects feature different types of risks, different levels of complexity requiring different skills and team sizes. The engineering processes need to be adaptable to accommodate these different projects needs. It must be possible to plug or unplug parts of a process without breaking the overall structure.

Often inside a Business Unit, different business domains cohabit. For instance, a Business Unit develops at the same time a maritime patrol system and a weapon navigation system. Both systems are mainly Control and Command, but underlying concepts, platforms and technologies are different. Development process is a way to capitalize and to spread development know-how. Processes must be generic enough to be shared and capitalized at company or department level, but must be customizable enough to be relevant at project level.

Processes describe roles, activities and work products for each involved stakeholder. The process work products can be documentation, models or source code. A number of tools are deployed to support the implementation of engineering processes. Each tool covers specific aspects (requirements capture, architecture structure description, traceability information capture, …) of one or more activity. Processes are interrelated activities, and then tools must communicate and share information to ensure process continuity. Systems engineering requires seamless tool chains that fully support the continuity of activities. Tools are used to ensure productivity and quality, which means that tool chains are on the critical path of many projects. Tooled-up processes[1] are assembled and managed by dedicated Tools & Methods organisations in THALES.

Processes have a long life cycle comparing to tools and technologies. In a process definition, activities, concerns and roles are the most stable parts. Methodological and technological choices have a shorter life cycle but tools (either commercial or in-house) have the shortest. To be able to master and limit impacts of evolutions, each of the previous aspects must be as much as possible independent of others. For instance, if technological choices like “UML as modelling language” have been made, UML profiles and modelling rules should be as much as possible independent from a specific UML modeller.

A number of THALES Units are currently involved in transitioning parts of their processes from document-driven to model-driven engineering (MDE). MDE puts models on the critical path of system and software development, turning the role of model from contemplative to productive. Putting MDE into practice in operational contexts intrinsically puts heavy demands on the tooled-up process: modelling rules and practices need to be precisely defined for each model work product; model edition, analysis, transformation and management require dedicated tools or tool extensions at each step of the process.

A first challenge is to support MDE tooled-up process definition, implementation, assembly, and deployment all the while addressing industrial concerns for adaptability, maintainability and scalability.

A second challenge is to be able to introduce MDE innovations in existing development processes at mastered cost, all the while managing the legacy. While some business contexts can allow for in-depth process evolution, more often a process is in place, which includes some level of modelling, and only limited evolutions can be afforded with respect to cost / benefit, project schedule constraints, legacy management constraints, etc.

The MIRROR Pilot Program was launched three years ago in THALES to address these challenges. MIRROR is addressing core issues for the definition and adoption of MDE approaches in the THALES group, including methodological, technological and tooling concerns [5].

The concept of MDE methodological component was identified in MIRROR as a building block for supporting the definition and building of MDE tooled-up proc-

[1] A “tooled-up process” is the union of a process and its supporting tool chain.

esses. An implementation approach for this concept is developed here within the context of a THALES industrial case, based on an extension of the IEEE Recommended practice for architectural description of software-intensive systems, IEEE Std 1471-2000.

This paper is organised as follows. Section 2 recalls the requirements in building a MDE process, introduces the concept of MDE methodological component and describes the extensions made to IEEE Std 1471-2000 to support the specification of UML–based viewpoints in a MDE methodological component. Section 3 presents the application that was performed of this approach in the context of the THALES Airborne Systems Unit. Section 4 discusses the insight retrieved from this experiment with regards to the different challenges THALES Business Units are facing for introducing MDE innovations in existing industrial development processes. Section 5 lists a number of perspectives regarding future MIRROR work on MDE methodological components for defining and assembling tooled-up processes.

2 Extending IEEE 1471 for MDE Processes

A Model-Driven Engineering process organises engineering activities around the building and exploitation of a number of inter-related model work products representing different aspects of the system under development, at different levels of abstraction. Engineering activities involve the building, analysis and transformation of models of the system, and the generation of code, configuration file, tests out of the models.

Building a MDE process requires:

- The definition of the model work products. A model work product uses a precisely defined modelling language, with precisely defined modelling rules; this language may be standard or specific to a company, a domain, a method, a technology. Model work products are set in a global frame with well-defined consistency relationships between models.
- The definition of transformation rules between given sets of models, of rules and procedures for generating code or documents from the models. These rules embody development know-how at domain or technological level.
- The identification of verification rules and techniques upon these models (functional, performance, quality verification).
- The integration of dedicated tools and tool extensions to support the model edition, verification, transformation, configuration management and generation practices in a cost-effective, industrial way.
- The definition of the engineering activities, practices and roles with regards to the model work products, the overall organisation and integration of the MDE tooled-up process, and the documentation of the process.

The concept of MDE methodological component was identified in MIRROR as a building block for supporting this complex MDE process-building task. This concept is presented hereafter; an implementation approach for this concept is then developed, based on an extension of the IEEE-1471 standard.

2.1 The Concept of MDE Methodological Component

A structural unit, the MDE methodological component [2,7], has been identified by MIRROR to be used as a vector for defining, implementing and deploying MDE methodologies. MDE Components are defined as containers for coherent and self-defined domain, technological, or methodological know-how. These containers may serve as building blocks for building complete and coherent tooled-up processes.
A MDE methodological component is a unit for[2]:

1. **Defining and Documenting a MDE Process:** A MDE component can hold the specification of a modelling language and organisational rules for a work product; the specification of a model transformation; the specification of verification rules; the specification of a process or sub-process in terms of activities, roles and work products; guidelines. Specification means were defined in MIRROR for each type of information; a modeling language is typically defined as a MOF[3] model for abstract syntax and a UML profile for concrete syntax; the UML SPEM[4] Profile is used for process specification.
2. **Assembling and adapting a MDE process for a project:** MDE components are modular process assets that can be assembled to build a complete process; specific extension mechanisms are defined to enable adaptation to contexts.
3. **Developing and managing tool support for a MDE process:** A MDE component holds information that can be exploited at tool-level, for implementing the process, enhancing model edition productivity and supporting the automation of some transformation and verification activities. This involves the generation of modeller extensions such as edition accelerators, well-formed ness rules checkers, quality analysis scripts, transformation scripts, documentation or code generator, etc.

Several granularities are foreseen for a MDE Component, including:

- A standalone **definition of a model work product**: modeling language, consistency rules, guidelines supporting related modeling activities, etc.
- A complete **development process** that assembles a set of MDE Component into a full MDE chain.

The MDE methodological component concept is designed to fulfill process needs in term of tooled-up, pluggable, and deployable, “development know-how” units. MIRROR has been experimenting several approaches to the definition and deployment of MDE components. This paper focuses on the usage of the IEEE Recommended practice for architectural description of software-intensive systems, IEEE Std 1471-2000 [1] as a MDE component specification support.

2.2 IEEE 1471 Recommendation

The THALES development processes are largely “architecture centric”; mastering product architecture has been identified as a key practice for mastering the overall

[2] In this paper we will mainly focus on aspect 1 and 2 that are pre requisite for aspect 3.
[3] Meta-Object Facility.
[4] Software Process Engineering Metamodel.

product. The definition of architectural viewpoints on a system is precisely the topic of the IEEE 1471.

IEEE 1471 is the Recommended Practice for Architectural Description developed by the IEEE's Architecture Working Group. IEEE set the following goals to the standard:

1. To take a wide scope interpretation of Architecture applicable to software intensive systems.
2. To establish a conceptual framework for talking about architecture issues.
3. To identify and promulgate architectural best practices.

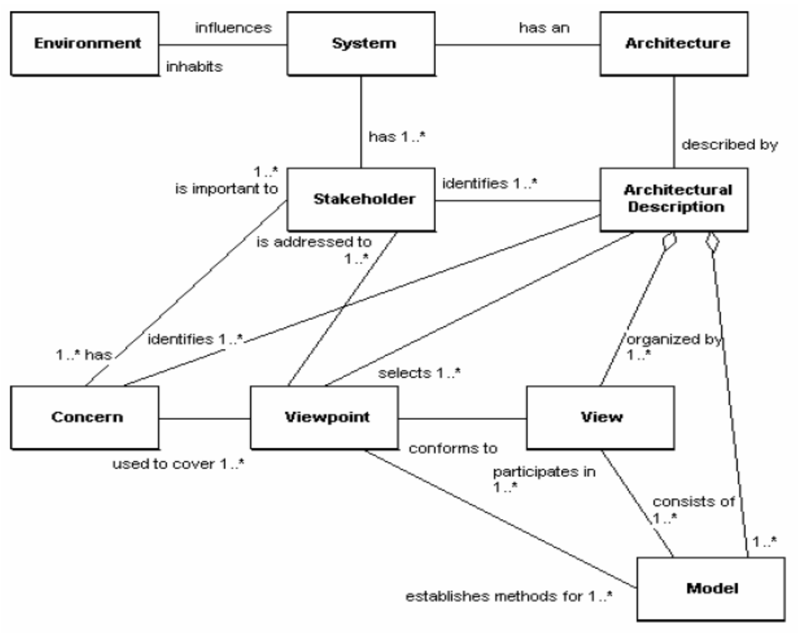


Fig. 1. IEEE 1471 conceptual model

Figure 1 above represents the conceptual model proposed in IEEE 1471. This model is articulated around two key concepts: view and viewpoint.

- **View:** "A representation of a whole system from the perspective of a related set of concerns."
- **Viewpoint:** "A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis."

"Like a legend on a map or chart, a viewpoint provides a guide for interpreting and using a view, and appears in a conforming AD together with the view it defines."[3]

A viewpoint is seen as a reusable asset that can be shared between projects and teams. Each viewpoint embeds its language, its modelling rules, its analysis methods, its completeness and coherency checking rules, and its guidelines.

Interestingly enough, [3] notes that: “It would be useful to have a standard way to document viewpoint declarations in the UML, such that they may be notationally depicted, stored and manipulated by tools.” This concern clearly matches the MIRROR concern that led to the concept of MDE methodological component outlined above. Our approach here is to base the work product specification part of our MDE components upon the IEEE 1471 Viewpoint concept, and to develop UML specification means for this Viewpoint concept.

The following section describes how the IEEE 1471 concepts were extended by MIRROR to support the specification of UML–based viewpoints.

2.3 Extending the IEEE 1471 Concepts

Hereafter represents the extended conceptual model for supporting UML-based Viewpoints. The main extended concepts are:

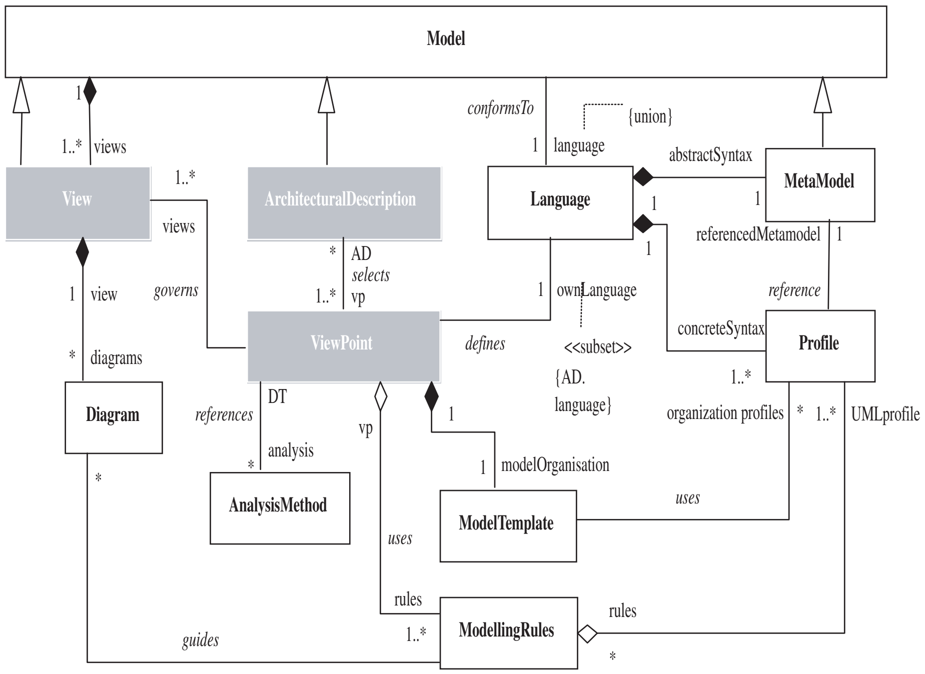


Fig. 2. Extended IEEE1471-MDE conceptual model[5]

[5] Grey boxes are original concepts coming from [1] while white boxes are concepts that extend IEEE 1471. Parts of the white boxes are based on [3].

- **Model:** A representation of a part of the function, structure and/or behaviour of a system. A model is composed of different views and conforms to a meta-model.
- **Language:** concepts and concrete notation that is used in the viewpoint.
- **Meta-model:** A MOF model describing concepts of a domain, technique or technology. It is the abstract syntax of the viewpoint's language.
- **UML Profile:** Set of UML extensions (stereotypes and tagged values) that is viewpoint's meta-model mapping on UML. UML profile is the concrete syntax of the viewpoint's language.
- **Architectural Description:** Architectural description is extended from [1]. It becomes a model. The meta-model of an Architectural description is then the union of the meta-models of its viewpoints.
- **Viewpoint:** Viewpoint is extended to make explicit the notion of viewpoint language as a MOF model and a UML profile. Viewpoint also embeds a "model organisation template" which is a way to describe how concepts will be stored in the model. The viewpoint also provides modelling rules.
- **Modelling Rules:** A set of rules that describes the usage of the profile. Modelling rules are a way to identify and specify contents and graphical rules of required UML diagrams.

The following section describes how we apply the extended IEEE 1471 Viewpoint to support the specification of MDE methodological components in an industrial THALES context.

3 Application

The case concerns THALES Airborne System (TAS), a THALES Business Unit operating in the Avionics business domain. TAS has a strong background in using modelling and object-oriented technology, with a UML tooled-up process for software engineering deployed in Rafale and Mirage contexts since 1998. This process is regularly enhanced and improved through consolidations and innovation introduction. [4]

The authors were actively involved in the identification and definition of MDE innovations for the TAS development processes.

Process innovations are managed through an iterative and incremental process:

1. **Legacy Analysis:** During this phase, existing processes are analysed and potential improvement areas are identified.
2. **Solution Design and Experimentation:** For a given improvement area, solutions are investigated, prototyped and experimented; impacts are identified, and a cost/benefit analysis is conducted as part of the effort.
3. **Decision:** Results are then submitted to a Steering Committee which decides to deploy or not.

3.1 Analysis Phase

The studied development process revolves around the production of a UML model that can be characterised as a « high-level design model », a hybrid model answering architectural concerns and partial code generation objectives.

TAS uses a Component-Based Development approach for its architecture design. Three abstraction levels have been identified:

- A coarse grain level that can be seen as system level.
- A medium grain level that can be seen as software component level.
- A fine grain level that can be seen as software classes level.

A specific UML profile is used to capture technical concepts.
Users rely on only two guidelines for mastering their activities and work products:

- **A design recommendations guideline:** Here designers find a full glossary of all the domain and technical concepts, they also find modelling rules and the description of the activities. Per activity, they find the modelling steps they need to follow.
- **A model organization guideline:** Here designers find how the model is structured in term of UML packages and where each type of model element must be stored in the UML Case tool.

Regarding process definition, the analysis resulted in the identification of a number of limitations in the UML model (and the underlying engineering activities) regarding the capture of structural, functional and behavioural aspects of the design.

Regarding process documentation material, the conclusions were that if required information was available, it was vague and distributed in different parts of the two documents. Therefore, it is hard to know when and where specific aspects must be produced.

3.2 Design and Experimentation Phase

Our approach was to introduce modularity in the definition of the TAS work products through defining a MDE component for each aspect of the design model work product under consolidation. Each MDE component was defined as an IEEE 1471 Viewpoint along the principles presented in section 2.

The following items are mandatory to fully define a MDE Component, which contains all the required information for a Viewpoint definition. A document template was defined for Viewpoint specification, containing the following sections:

1. **Concerns:** What is (are) the Viewpoint objective(s)?
2. **Roles and Activities:** Definition of Process activities for building the View. As much as possible, the modelling language for process description should be the SPEM UML profile, otherwise a textual description is allowed.
3. **Language:** Definition of the Viewpoint meta-model (concepts) and of the UML profile[6] (concrete notation) used for the Viewpoint.
4. **Modelling Rules:** Modelling guidance (how to use the profile for specific purpose) is provided in natural language.
5. **Model organisation:** Definition of how the model is structured in terms of packages and where model elements must be stored.
6. **Rationale:** Definition of how the modelling and storing choices aim at fulfilling viewpoint's objectives and needs.

[6] Mapping the Viewpoint meta-model to UML results in the UML profile.

7. **Viewpoint dependencies:** A Viewpoint covers a subset of engineering concerns. Covering all concerns means building a coherent set of viewpoints. Often, some viewpoint will depend on another. By making the dependencies explicit, we allow process designers to master their kind of "Viewpoints architecture".
8. **Analysis methods:** Specification of the methods for analysing and verifying the model, e.g. for quality, well-formed ness, performance.

A number of Viewpoints were identified in the TAS context, both within the scope of the existing modelling practices and for extending these practices to cover the identified limitations. These Viewpoints are structuring vectors for both consolidating existing modelling practices and introducing MDE innovations. The identified Viewpoints are listed below:

- **Structural Viewpoint:** The core Viewpoint on which most of the other Viewpoints will be based. In this Viewpoint, essential architectural elements and relations are described along a component-based approach, using different concepts depending on the abstraction level (system, component, class, interface, port, service).
- **Delegation Viewpoint:** The Structural Viewpoint defines components with interfaces that offer services. This Viewpoint addresses the definition of interface services realization across the different abstraction levels (from coarse grain components to classes).
- **Realization Viewpoint:** All architectural elements must collaborate to at least one functional or non-functional requirement. This Viewpoint specifies the collaboration of architectural elements for requirement realization and the related behavioural scenarios.
- **Performance Viewpoint:** In the real-time embedded domain, performance and, more generally, technical dimensioning concerns are critical. This Viewpoint addresses the specification and verification of performances features using UML techniques.
- **Scheduling Viewpoint:** This Viewpoint addresses the scheduling of the system computations, in relation to the features of the underlying real-time multi-task platform. This viewpoint is critical.

Structure, Realization, and Delegation viewpoints were produced and experimented for all abstraction levels as a priority. The experimentation led to several iterations before obtaining satisfactory results validated by reviews including both end users and Method and tools stakeholders.

3.3 Decision Phase

Each new or consolidated Viewpoint was presented to the process innovation Steering Committee along with an analysis of its impact, costs and expected benefits. Structural and Delegation viewpoint innovations were accepted for short-term deployment in a business context; the deployment of other innovations was delayed due to cost or project schedule constraints.

4 Discussion

Our extended Viewpoint concept has proven a valuable tool for specifying in a modular way UML model work products in a MDE process.

Our Viewpoints are self-contained and coherent units that fully specify the modeling rules and practices for addressing a given engineering concern, at a given step of the process, within a given model work product. Viewpoint dependencies are managed explicitly, allowing the process engineer to master a kind of “viewpoints architecture”, controlling Viewpoint assembly and modification impacts.

The Viewpoint “template” is a valuable tool for unifying tooled-up process definition at company or department level, and for building capitalized MDE process assets. A common Viewpoint typology can be defined, and a Viewpoint library can be developed. In the TAS context, the set of Viewpoints was first defined for the Weapon navigation systems department; these viewpoints are currently adapted to fit the needs and practices of the Maritime patrol system department.

The enabled modularity, based on the separation of engineering concerns, facilitates the management of the overall MDE process definition complexity for the method designer, but also for the tool developer, for the user of the method and for the manager in charge with deployment decision.

For the tool developer, the Viewpoint is a unit for managing process tooling requirements, and for organizing tool facilities that can be assembled and deployed in the tool chain for a MDE tooled-up process. For example, the TAS Delegation viewpoint is used as an input for the specification of a dedicated set of helpers developed as plug-able extensions to the UML modeler tool.

For the user of the method, a presentation of modeling practices along engineering concerns may be easier to exploit, with regards to monolithic specifications.

This extended Viewpoint also seems a meaningful unit for MDE innovations introduction in an engineering process. It federates in a unique repository all the stakeholders required information. Articulating modelling practices around modular Viewpoints focused on engineering concerns allows managers to define priorities and to incrementally deploy innovations, on a Viewpoint basis. The managers are provided a clear view on the targeted engineering concerns, and impacts on legacy, cost and benefits can be assessed separately.

5 Perspectives

The extended Viewpoint concept that was presented in this paper and applied in the operational TAS context provides the basis for MDE methodological component as defined in section 2.1. It takes the form of (meta) models and textual documents for MDE process definition and documentation, and for process innovation management. MIRROR is currently taking MDE components one-step further through developing approaches for turning meta-models and profile constituents into productive assets for tooled-up processes.

Meta-models are first-class constituents of MDE components. MIRROR is currently investigating the exploitation of the meta-models and profiles to automate different aspects of the production of tooled-up processes:

- **Generating the methodological documentation of described concepts:** Concepts constitute the language that will be shared between all stakeholders; they are carefully modeled and documented. The meta-model is a concept repository that can be used for generating different kinds of documents (training, glossaries, process guidelines), alleviating the costs of document maintenance.
- **Generating modeling tool extensions for syntactical checking of the model:** UML tools have built-in facilities for syntactical checks on basic UML; the modeling language defined for a given model in a MDE process defines specific modeling rules the checking of which requires dedicated tool extensions. These rules are fully defined in the meta-model (concepts, relations, cardinalities and constraints). Scripts for the syntactical checking of models can be generated out of the meta-model information, for a given UML profile, and for a target UML modeler, thus decoupling rule specification and UML modeler-specific script implementation.
- **Generating productivity enhancement tool extensions:** Using a UML profile in a basic UML modeler often requires many costly tool manipulations for the end user to set the proper stereotypes and tagged values. Users have to work at UML level rather than at domain concepts level. Dedicated GUIs can be developed to enable edition at this domain level; these wizards hide the UML details through providing domain-level edition facilities. A large part of these wizards can be deduced from the meta-model information. MIRROR is exploring the usage of the meta-model to generate GUI or contextual menus, decoupling concepts definition from specific UML tool and specific UML profile usage.

Along with these automation perspectives, MIRROR is developing the different dimensions of MDE methodological component storage, composition, versioning, and deployment.

Last but not least, MDE components are work products of process assets development. Then, a methodology for developing MDE component is mandatory to ensure productivity and quality. We need a tooled-up process for tooled-up process definition.

Acknowledgement

We thank all Thales Airborne Systems members, especially Frédéric Maraux, Laureline Vagner, François Pennaneac'h, Patrice Colzatto, and Bernard Kerbiquet ,for their contribution. We also thank all MIRROR project members, especially Benoit Langlois, for their contribution and the time they spent in reviews.

References

[1] IEEE Architecture Working Group, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems,IEEE Std 1471-2000, IEEE, 2000

[2] MDA Components: Challenges and opportunity J.Bezivin, S.Gerard, P-A. Muller, L.Rioux Metamodelling for MDA 24-25th November 2003 York

[3] Using the UML for Architectural Description. In Proceeding of <>'99 The Unified Modeling Language: Beyond the Standard. 2nd International Conference, pages 32--48, 1999

[4] THALES Systèmes Aéroportés : L'approche processus avec UML. J.Bargallo, F Maraux. 2e conférence annuelle d'ingénierie système AFIS. Toulouse 2001

[5] Practicall Experiences in the Application of MDA. M de Miguel, J Jourdan, S Salicki In Proceeding of "UML" 2002 The unified modelling language. 5th International conference. P128-139, 2002

[6] OMG. Model Driven Architecture (MDA) Guide v1.0.1 Document number ab/2003-06-01, June 12, 2003.

[7] P. Desfray, "When a Major Software Trend Meets our Toolset, Implemented since 1994, Nov 2001 - http://www.omg.org/mda/presentations.htm

Data Communications Standards: A Case for the UML

Otto Preiss, Tatjana Kostic, and Christian Frei

ABB Switzerland, Corporate Research,
5405 Baden-Daettwil, Switzerland
{otto.preiss, tatjana.kostic, christian.frei}@ch.abb.com
http://www.abb.com

Abstract. This paper argues for the UML as a means to rigorously specify data models and communications services of industrial data communications standards. Such standards contain among others the syntactic and semantic description of the application data to be exchanged among devices and systems of different vendors. As a consequence of the growing number and complexity of types of application data and their resulting specific communications services, it becomes almost impossible to maintain specification consistency and avoid ambiguities on the basis of informal, textual notations. By means of example, this paper discusses the recently accepted IEC 61850 standard "Communications networks and systems in substations" and its shortcomings due to the lack of formal notations. It is shown how many of these shortcomings can be overcome and additional benefits provided through the use of UML models – models that could ultimately find their way into the normative parts of such standards.

1 Introduction

Industrial automation is largely based on distributed systems, which increasingly have to integrate products and applications of different vendors. In order to do so, many domain-specific data communications standards (often called "protocol standards") are defined. This can be for domains such as manufacturing control, automotive control, or control of substations and equipment which are part of our electric power systems.

In the past, protocol standards were rather simple. That is, they had a relatively small number of data types defined and also a small number of types of application communications services. Further, the data types were comparably simple and close to what software programmers would call primitive data types. The mapping of the application data items to underlying communications stacks was also rather straightforward, not at last because the standards usually defined all or the required layers[1] of a communications stack on their own.

This simplicity, however, has turned into complexity. The complexity has a number of reasons. (a) Structurally much more complex pieces of information shall be exchanged. (b) More elaborate communications services are realized through the adoption of communications techniques and service models that were introduced in

[1] Layers in the sense of the 7-layer model of the ISO/IEC 7498 [1].

standard information technology (IT). For example, this includes communications techniques based on confirmed and unconfirmed services, elaborate event models, security services, transaction support, remote procedure calls, etc. (c) It is desirable that the communications standards can make use of existing communications layers or profiles (such as Ethernet for physical and link layer, IP and TCP for network and transport layer, etc.). However, these more complex pieces of information and more complex types of services are still constrained by the soft real-time nature of the industrial control applications. As a result, the mix of domain-specific requirements with the reuse of standard IT approaches and technologies leads to big, complex standards with many implicit and explicit dependencies among parts of such a standard.

In this paper, we argue that while in the past it was possible to specify and maintain data models of protocol standards based on informal, textual notations, it is now simply impossible to do so. We support our arguments by comparing the "prose text and table" approach of the IEC 61850 with a rigorous specification using the UML. During our work, which consisted of the formalization of the IEC 61850 data and service model, we identified numerous existing inconsistencies in the standard. We also learned to appreciate the availability of a UML model during the development of software that is in one or the other way using the IEC 61850 standard. As a further motivation for UML models, we should not forget that it is the software developers who need to understand the specifications written mainly by substation domain experts. Hence, to avoid the many possible interpretations of software developers, a common language for specification should be chosen that facilitates the developers tasks.

The rest of the paper is structured as follows: Sections 2 and 3 briefly introduce the application domain and the IEC 61850, respectively. By means of one concrete example, Section 4 compares the specification of the IEC 61850 application data model "as-is", i.e. through informal text and tables, with a UML-based specification. Section 5 lists the specific advantages of the UML approach, before Section 6 concludes with a short summary and outlook.

2 Application Domain

Electric power networks (power grids) are vast systems that are responsible for transporting energy from generation sites to end-consumers such as individual households or larger industries. The nodes in such a network are called substations and take over the voltage transformation and/or the routing of energy flow by means of the installed switchgear (e.g. transformers, circuit breakers). Substations are controlled by Substation Automation Systems (SAS), which are composed of all the electronic equipment that continuously control, monitor, and protect the network and its high voltage equipment, so as to avoid unplanned electricity outages.

An SAS can be classified as a distributed soft real-time system and as such is one possible type of data acquisition and process control system. It is usually composed of 20 to 100 Intelligent Electronic Devices (IED) which are connected by a high-speed communications network with possibly routers and gateways. Any device that can run executable code and provides a data communications interface would classify as an IED (e.g. an intelligent sensor, an embedded device, a programmable controller, a workstation PC). While some real-time critical functions are executed more or less autonomously on a single IED, other functions are realized in distributed form over many IEDs. However, even the functions running on a single IED are expected to communicate with the outside world, e.g. to provide some status information to operators.

A communications standard is required to be able to exchange application data among IEDs, particularly among IEDs from different vendors, in a predefined way. This is needed to prevent costly, project-specific hardware and software developments. The IEC 61850, discussed in the next section and called *the standard* in the sequel, is the first comprehensive and recently accepted standard in the Substation Automation (SA) domain.

3 The IEC 61850 Standard: Communication Networks and Systems in Substations

The IEC 61850 set of documents [2] is divided into ten parts (Part 1 - Part 10) but content-wise circles around five major topics, where all but the functional model have a normative character:

- Functional modeling: a *functional model* of the SA domain is conceived, but is mainly used to derive the quality of service requirements on the communications system (standard's Part 5). However, it implicitly documents the authors' thinking, i.e. their conceptualization of the relevant aspects of the SA domain.
- Data modeling: a *data model* for SAS is defined. It defines the syntax (naming scheme) and semantics of the application level data that can be exchanged in SA systems (Parts 7-2, 7-3 and 7-4).
- Communication service modeling: a *communications service model* defines the different ways of accessing data of IEDs (Part 7-2).
- Communications protocol stacks: the communication services and data models are mapped to concrete *data communications protocol stacks* (Parts 8 and 9).
- SAS engineering and testing: an XML based *Substation Configuration Language* (SCL, Part 6) is defined to describe the substation and the automation system topology, as well as the communications–relevant configuration data of IEDs. Part 10 describes the recommended conformance testing for IEC 61850 compliance.

The standard implicitly introduced a terminology that we conceptualized and explain with the help of Fig. 1 and Fig. 2. We will only briefly visit those concepts that are relevant to the discussion in this paper. A more detailed discussion is available in [3] and, of course, in the standard itself.

The functionality of a substation automation system is described by a "logical system" that is comprised of the set of functions that operate in the substation environment. This is seen on the left branch of the conceptual model in Fig. 1[2]. Functions can be thought of as high-level system operations, e.g. "open a high voltage switch (called breaker) to de-energize an overhead line". Functions are conceptually realized by (collaborations of) primitive, atomic functional building blocks, the so-called *Logical*

[2] The role annotations of associations are used to support the reading of the conceptual diagram, e.g. a Function is "+distributed over" 1…n Physical Devices.

Nodes (LN). The standard identifies some 90 predefined types of logical nodes in Part 7-4. The physical configuration of an SAS is modelled as a distributed system of interconnected devices, typically IEDs (right branch of Fig. 1). A certain SAS function is considered distributed, if it is realized through the deployment of its collaborating LNs to two or more IEDs (see also Fig. 2, where Function XYZ is realized by four LNs). LNs are logically connected by the concept of a Logical Connection (LC). In software architectural terms, an LC represents the connector between LNs, i.e. it models a communication association and thus abstracts the protocol of interaction between two LNs [4]. The abstract communications services are defined for LNs (Part 7-2) through the envisioned types of LCs.

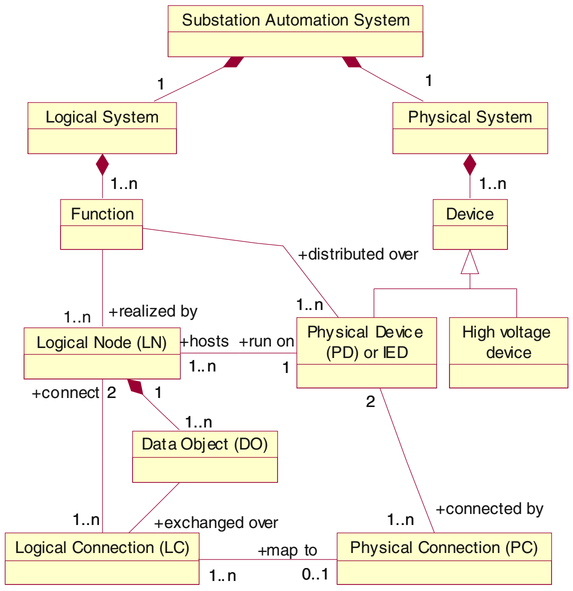


Fig. 1. Conceptual model of the key terms used in the IEC 61850

The application data, which is exchanged over LCs through the defined types of communications services, is modeled and described by the concept of a *Data Object* (DO). Hence, it is the LNs that have abstract communications services specified and contain Data Objects, which they can make available as part of a data exchange. A Data Object defines the structure and the semantics of the exchanged data items in the form of a well-specified, domain specific, abstract data type. The standard defines about 30 specific types of DOs. These predefined types are generally called *Common Data Classes* (CDC) and are described in Part 7-3.

For easier understanding, Fig. 2 depicts an instance diagram with a fictitious usage of the concepts as shown in Fig. 1. Theoretically, the components of the logical system have no predefined allocation to the components of the physical system, i.e. LNs may be distributed in any way and thus are not bound to certain types of IEDs. Practically however, the performance-related requirements together with the limited physical resources (network bandwidth, processing power) constrain the number of feasible mapping scenarios.

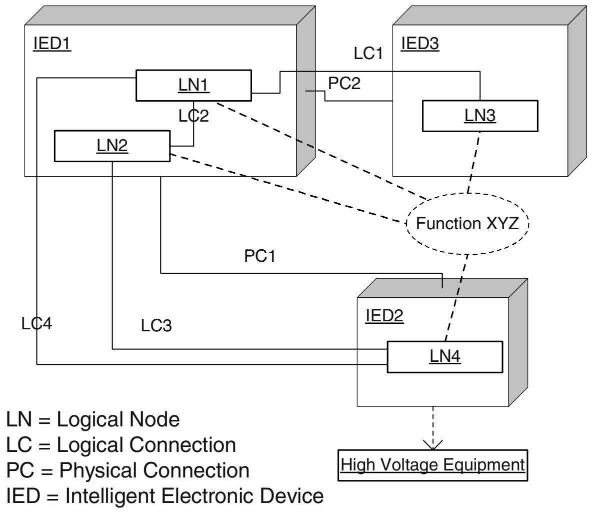


Fig. 2. An SA function is realized by a set of collaborating LNs distributed over IEDs

4 Text-and-Table Versus a UML Model: An Example

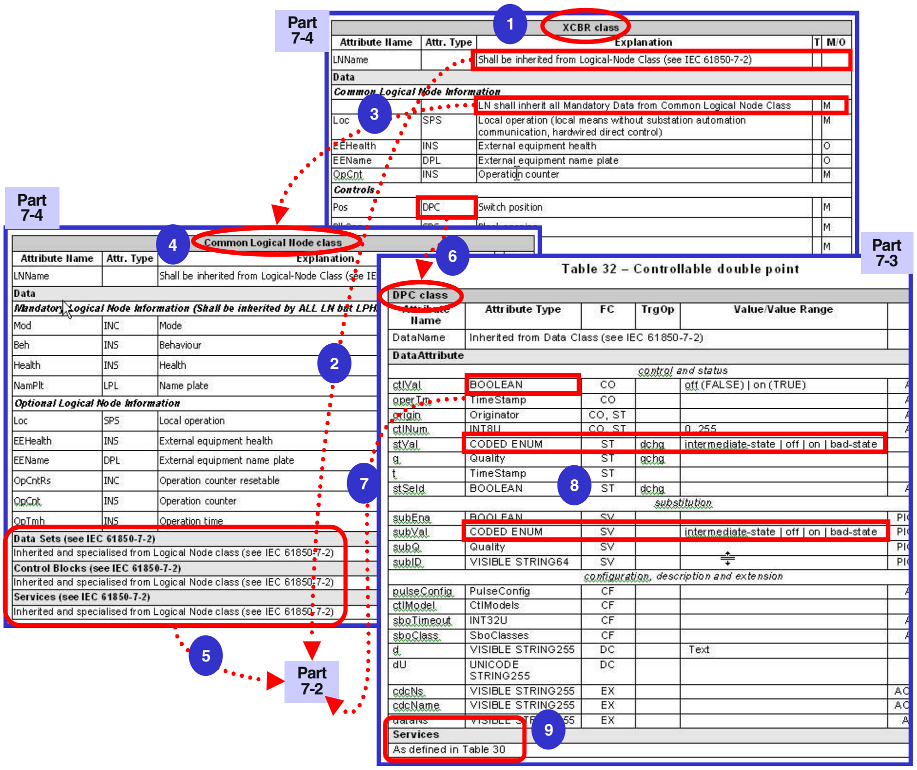
The previous section introduced two key concepts: Logical Nodes (LNs), which contain Data Objects, and Common Data Classes (CDCs), which are defined types of Data Objects. CDCs represent substation domain specific, complex data types, which are composed of other complex or primitive data types. Both LNs and CDCs have abstract services defined that enable access to their data contents down to the level of their primitive attributes. These three fundamental concepts (LNs, CDCs and services) are specified throughout the Parts 7-4, 7-3 and 7-2 of the standard, respectively.

In this section, we illustrate with an example of one concrete LN, called `XCBR`, how this LN is actually defined in the standard and how we have modelled it in the UML. `XCBR` is a type of an LN that represents a circuit breaker. The latter is a high voltage current interrupting device (think of it as a "very big" light switch). It can be used to energize/de-energise ("switch on/off") a high voltage line feeding a big city.

In Section 4.1, we first show how the `XCBR` is defined in the standard and what the procedure is to find the relevant, normatively defined, parts of its specification. In Section 4.2, we then show and discuss the formal UML model for this same `XCBR`.

4.1 Text-and-Table Definition of the Logical Node **`XCBR`**

Fig. 3 illustrates a part of the definition of the `XCBR` Logical Node, in the form of tables, which are spread throughout different documents (i.e. throughout different parts) of the standard. For easier reference in the following discussion, we have added sequence numbers (filled numbered circles) in Fig. 3 to those points that are being emphasised.



Part 7-4
1
2
3
Part 7-4
4
5
Part 7-2
6
7
8
9
Part 7-3

XCBR class				
Attribute Name	Attr. Type	Explanation	T	M/O
LNName		Shall be inherited from Logical-Node Class (see IEC 61850-7-2)		
Data				
Common Logical Node Information				
		LN shall inherit all Mandatory Data from Common Logical Node Class		M
Loc	SPS	Local operation (local means without substation automation communication, hardwired direct control)		M
EEHealth	INS	External equipment health		O
EEName	DPL	External equipment name plate		O
OpCnt	INS	Operation counter		M
Controls				
Pos	DPC	Switch position		M
				M
				M

Common Logical Node class		
Attribute Name	Attr. Type	Explanation
LNName		Shall be inherited from Logical-Node Class (see IE
Data		
Mandatory Logical Node Information (Shall be inherited by ALL LN but LPH		
Mod	INC	Mode
Beh	INS	Behaviour
Health	INS	Health
NamPlt	LPL	Name plate
Optional Logical Node Information		
Loc	SPS	Local operation
EEHealth	INS	External equipment health
EEName	DPL	External equipment name plate
OpCntRs	INC	Operation counter resetable
OpCnt	INS	Operation counter
OpTmh	INS	Operation time
Data Sets (see IEC 61850-7-2)		
Inherited and specialised from Logical Node class (see IEC 61850-7-2)		
Control Blocks (see IEC 61850-7-2)		
Inherited and specialised from Logical Node class (see IEC 61850-7-2)		
Services (see IEC 61850-7-2)		
Inherited and specialised from Logical Node class (see IEC 61850-7-2)		

Table 32 – Controllable double point

DPC class					
Attribute Name	Attribute Type	FC	TrgOp	Value/Value Range	
DataName	Inherited from Data Class (see IEC 61850-7-2)				
DataAttribute					
				control and status	
ctlVal	BOOLEAN	CO		off (FALSE) | on (TRUE)	A
operTm	TimeStamp	CO			A
origin	Originator	CO, ST			A
ctlNum	INT8U	CO, ST		0..255	A
stVal	CODED ENUM	ST	dchg	intermediate-state | off | on | bad-state	
q	Quality	ST	qchg		
t	TimeStamp	ST			
stSeld	BOOLEAN	ST	dchg		A
				substitution	
subEna	BOOLEAN	SV			PI
subVal	CODED ENUM	SV		intermediate-state | off | on | bad-state	PI
subQ	Quality	SV			PI
subID	VISIBLE STRING64	SV			PI
				configuration, description and extension	
pulseConfig	PulseConfig	CF			A
ctlModel	CtlModels	CF			
sboTimeout	INT32U	CF			A
sboClass	SboClasses	CF			A
d	VISIBLE STRING255	DC		Text	
dU	UNICODE STRING255	DC			
cdcNs	VISIBLE STRING255	EX			AC
cdcName	VISIBLE STRING255	EX			AC
dataNs	VISIBLE STR[illegible]55	EX			A
Services					
As defined in Table 30					

Fig. 3. An excerpt of the XCBR Logical Node definition from three parts of the standard

The main table for XCBR (circle number 1) is defined in Part 7-4 and it is the place to start. It specifies the attributes of an XCBR, i.e. their names and types, a textual explanation, and their cardinalities (mandatory or optional in the column "M/O"[3]). The first attribute of XCBR is its name, LNName. It is said to "be inherited from Logical-Node Class", which is defined in Part 7-2 (circle 2). This "inheritance" refers to a definition of a generic structure for any LN, including the definition of the name attribute. We omit further details to not make matters more complicated. The list of Data Objects contained in an XCBR is referred to as the "Data" section in the main table. Individual Data Objects are listed by and distinguished through their "Attribute Name". The first, but empty (i.e. not named and not typed), attribute/Data Object stands for "all Mandatory Data from Common Logical Node Class". This informally points to another table (3) in the same part of the standard, which defines the so-called

[3] The column "T" is out of scope of this discussion.

Common Logical Node class (4), a kind of a root type LN. According to the explanation in this table, one can conclude that the first empty attribute of XCBR stands for the four mandatory Data Objects in table (4): Mode, Beh, Health and NamPlt. The next 6 Data Objects are optional in this table, but their usage in concrete LNs is defined in the standard as follows: "These optional data can be (a) inherited as mandatory, (b) inherited as optional, or (c) not inherited at all." In the case of XCBR, two of these attributes are inherited as mandatory (Loc, OpCnt), two as optional (EE-Health, EEName), and the remaining two are not inherited at all. This same table (4) specifies other concepts common to all LNs, including services, which are defined again in another document, namely in Part 7-2 of the standard (5).

Back to the main table for XCBR, we focus now on one Data Object: Pos. Pos stands for *position* ("switch on or/off"). Its type is that of a DPC. The latter is one of more than thirty CDCs defined in Part 7-3 of the standard (6). The structure of a DPC, and of any other CDC, is somewhat different from that of LNs, but we do not enter into those details here. Based on the example of the DPC, we only want to emphasize some points, which apply to any CDC in general.

The attribute names of a DPC start with an initial lower case letter, in contrast to attributes of LNs, which start with an upper case letter. Attributes of a DPC may be of either composite or primitive type. Primitive types are defined again in another part of the standard. In the example of a DPC, some of its attributes (e.g. "ctlVal") are of type BOOLEAN, which is again specified (7) in Part 7-2 along with other primitive (and some composite) types.

Finally, the services applicable to a DPC are defined in still another table (9) within the same document.

If the above description was hard to follow, we should emphasise again that the example was not made up by us (it even represents a rather simple case) but simply reflects how the standard is currently written and released. It goes without words that room for interpretation and human error is ample.

Note, as an example of a human error even in the standards definition, consider the bold frames around the definitions for the stVal and subVal attributes of DPC (8). It has gone unnoticed that the CODED_ENUM attribute type has no explicit type definition at all. Nevertheless is it "copy-pasted" from table to table and appears in several CDCs without any further definition as to how its semantics (intermediate-state|off|...) is represented syntactically.

4.2 UML Model of XCBR

The above short example should have given a feeling on how difficult it may be to grasp the high level concepts, and consequently to find ones way across the tables spread throughout the different parts of the standard. Added to the "navigation" complexity is the inconsistent nomenclature used, such as the meaning of inheritance, class vs. type, and Data vs. Data Object vs. attribute. Note, we have shown only one part of the definition of an XCBR Logical Node and of a DPC Common Data Class. Several more concepts have not even been mentioned. The current "as-is" specification is all but easy to unambiguously interpret by software engineers. But it is them who ultimately must implement software that is compliant with the standard.

Motivated by the need to understand the standard and to make it accessible to software engineers, we have developed a UML model [5] (in the sequel: *the UML model*), which formalises the data and the service model of the IEC 61850. Fig. 4 shows an extract of the UML model. It only illustrates the UML representation of those concepts that were discussed in the XCBR example of the previous section. We have again added sequence numbers to ease the discussion, and we did it in such a way that they correspond to the numbered topics from Fig. 3.

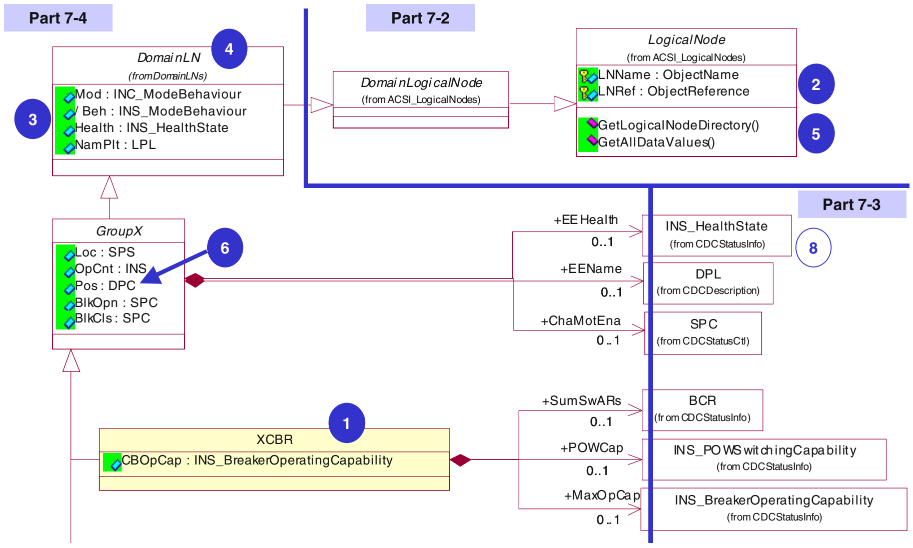


Fig. 4. UML class model of the Logical Node XCBR

The XCBR LN is a concrete type (1) that belongs to the inheritance hierarchy having as its root type the abstract type LogicalNode[4]. This root type has defined attributes (2) and defined services (5), which are common to all LNs (as specified in Part 7-2 of the standard). There are three basic types of LNs, but only the one relevant for XCBR, DomainLogicalNode, is shown in Fig. 4. For brevity, we did not even enter into this discussion in the previous section.

According to the definitions in Part 7-4, we model the Common Logical Node as a DomainLN (4), which groups the mandatory attributes (3) common to all concrete domain LNs. The standard defines XCBR within one of 13 groups of LNs, GroupX (again a finesse we have not mentioned in Section 4.1). This abstract supertype groups the attributes common to all the LNs within the group X.

In general, we model mandatory attributes, such as the Data Object Pos (6) in GroupX, as UML class attributes. Optional attributes are modelled as containment relationships of cardinality 0..1, where the role name is the name of the optional at-

[4] Note that we use the term type not in the restricted UML sense of a <<data type>> but in general as an object type. This permits the use of UML operations.

tribute. An example is `EEHealth`, defined as optional for all LNs from `GroupX`. Following this convention, we can see that the `XCBR` defines one mandatory and three optional attributes.

The types of LN attributes are the CDCs, in Fig. 4 either shown explicitly as class boxes or as type definitions of class attribute names, in which case the class boxes are omitted in the figure for brevity. As an example for the former consider the `GroupX`'s optional attribute `EEHealth`, whose type is `INS_HealthState` (8). As an example for typed class attributes, consider the previously discussed `DPC` type for the mandatory attribute `Pos` (6) of `GroupX`. All CDCs, which are defined in Part 7-3 of the standard, have their own detailed class representation.

If you recall our remark on undefined enumerations in Section 4.1, we should mention that the type `INS_HealthState` (8) is a CDC, which we introduced to circumvent this problem. We defined `INS_HealthState` as a specialisation of `INS` (the type as defined in the standard). In this specialized version, we explicitly incorporated the definition for the particular type of enumeration type. Hence, the (8) in the unfilled circle means that, opposed to the (8) in Fig. 3, the UML model explicitly defines enumerated types.

In Fig. 4, we do not show the detailed class definitions of CDCs such as `DPC` or `INS_HealthState`. Their structure is again based on inheritance and containment constructs and as such conceptually similar to that of the discussed XCBR. The detailed specifications are of course part of the full UML model [5].

5 Benefits of the UML-Based Approach

The example in the previous section has shown only a part of the definition of one type of Logical Node. The standard defines about 90 LNs, each having in average some 20 Data Objects. There are about 30 defined types (CDCs) of Data Objects. Each CDC has in average 20 attributes, half of them being again of some complex, composite type. The complex type definitions span across two more levels of containment. In addition to the LNs and their Data Objects, the standard defines a number of other aspects of LNs, which we have not discussed at all. All this may give an idea about the size and the complexity of the definitions in the standard. To give an intuition for the size, our UML model, which captures large parts of the current data model, consists of more than 900 classes with their respective associations.

In that context, it is easy to imagine that benefits arise in the development and the maintenance of the standard as well as in the use of it. In particular, we would like to stress the following points:

(1) **CASE tool support:** The rigorous UML model, maintained with a CASE tool, inherently provides consistency with respect to the defined "vocabulary" (concepts and concept relationships). Based on this, tools may then provide a wealth of features such as pop-up menus to select available types, support propagation of changes, etc. It is also easy to display for a specific type all the attributes (inherited or not), operations, associations and their cardinalities. This facilitates checking of the logical correctness of the type definition. The possibility to selectively display the model elements of interest on a class diagram allows one to create complex diagrams, such as

the full version of what is depicted in Fig. 4, while keeping their partial display manageable. CASE tools can usually provide complementary views of the model on the same screen. For instance, UML diagrams, the model browser, and the model element documentation may appear simultaneously. Consequently, it is easy to navigate between a UML model element in the browser and its representation in a class diagram, while still having informal, textual annotations of the selected element. Providing the same level of checking and comfort for the informal definitions based on text documents, even with an elaborate hyper-linking scheme, is almost impossible.

(2) **Model maintenance and extensions:** The formal model makes the model maintenance easier. It allows for consistent modifications from one version of the standard to another. For instance, if a type name is to be changed, it is done and potentially documented at one place within the model. This supports impact analysis and regression testing of planned modifications. Contrast this to the changes necessary at several places in at least three documents in the standard and the subsequent activities to assess further implications. A further argument for the UML model is the ease for programmers to extend the model with potential private, vendor specific extensions while preserving the compliance with the standard and its base model.

(3) **Code and documentation generation:** UML models can be used for code generation, which includes the direct use of inheritance relationships defined in the model. CASE tools also usually support roundtrip engineering. Both features facilitate the developers' tasks. Finally, the model documentation and, if needed, the code documentation can be automatically generated from the same model. For instance, with the use of advanced documentation plug-ins, we were able to reproduce the text and table look of the standard from the UML model. Moreover, other representations can be produced automatically, such as XML (with some limitations obviously) or RDF schemas for data serialization.

(4) **Formal data mappings or conversions to other standards:** The devices and applications, which implement the IEC 61850 specification, are not restricted to the substation automation domain. They have to inter-operate with systems and applications in the entire electric utility environment. In that sense, they do not only communicate with each other within a substation, but also with network control centres or even back-office applications. The UML model allows one to specify mappings to other existing UML models in a formal way. An example of such a mapping of the proposed UML model for the IEC 61850 was our mapping to CIM (Common Information Model). Details can be found in [6]. CIM specifies a data model as part of an IEC standard (IEC 61970 [7]) for electrical network control centres. Another mapping of the data model is that to lower communication layers. For instance, the UML model of the IEC 61850 should also facilitate a formal mapping to the MMS (Manufacturing Messaging Specification [8]). MMS is indeed one of defined application layer protocols defined in the IEC 61850. Hence, the standard defines appropriate communications mappings in Part 8-1, but again in text-and-table fashion.

Note that we might have achieved a similarly rigorous specification with the use of the SDL. SDL is a Specification and Description Language standardized as ITU (International Telecommunication Union) Recommendation Z.100 (for a collection of links to SDL material see [9]). However, the SDL has its strengths and emphasis clearly on system behaviour and structure modelling but less so on data modelling.

Further, SDL is less widely used and known by software engineers and also by substation domain specialists; in addition, its data part has no advantages over the UML. Hence, suggesting SDL to a community that is currently not using specification languages for data modelling at all would significantly lower the chances of being accepted.

6 Conclusion

We have illustrated with an example how an informal "text and table" definition of a data communications standard can be formalised using a UML-based representation. The benefits of such a formalisation are numerous and take advantage of the power of CASE tools. Such tools support easy navigation in the model, validation (typically, unknown classes will be identified immediately), and consistent maintenance (which is probably the hardest to meet with standards provided in text-and-table form). For instance, since 2002, when we started to model early drafts the IEC 61850 standard, we were able to identify many inconsistencies and problems. Over a hundred comments were sent to the IEC 61850 authors. Some of them were incorporated "as is" in the standard while others triggered discussions and clarifications. To give an intuition for the importance of consistency and un-ambiguity based on anecdotal evidence, we can mention that the first interoperability test between IEDs of different vendors was postponed after a week because, among others, there were too many (valid) interpretations that the different vendors took and implemented.

Besides validation support, software code (at least the skeleton, including the profile of the methods) can be generated automatically, thereby reducing the effort of developers and avoiding mistakes. Additionally, for those not familiar with UML CASE tools, text documentation can be generated automatically too, and be it in a stylish text-and-table format, if so expected by domain experts. Additional models, such as data serialization formats, can also be produced automatically. The UML then serves as the master model containing all definitions and comments. Further, a UML model can be used to specify mappings to other existing UML models (e.g. defined in related standards) in a formal way, and therefore can serve as a formal documentation for application integration.

As mentioned above, our findings were continuously submitted to the IEC, and the UML model was made known to IEC editors . By this, we hope that the UML model (or parts of it) is at least non-normatively included in the upcoming releases of the standard. In general, the consistency validation and maintenance of the standard's evolution would be greatly improved. We also hope that in the future, the standardisation bodies defining non-IT domain standards, such as the IEC 61850, will consider involving software experts in the specification process right from the beginning. They could largely support domain experts to express their domain knowledge in a more precise way, so that the primary users of the specifications, mainly software developers, are able to correctly interpret, design and implement products compliant to the specification.

Note that, besides some areas of telecommunication where the SDL is being used, the IEC 61970 standard [7] is pursuing this path: the whole data model (CIM) is being defined in UML and it is the UML model, with its annotations, that is going to be normative.

References

1. ISO/IEC 7498-1:1994: Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model
2. IEC 61850:2003: Communication networks and systems in substations, Part 1 to Part 10, International Electrotechnical Committee, Geneva, 2003
3. Preiss, O., Kostic, T., Frei, C., The major abstractions and models of IEC 61850, ABB Switzerland Ltd., Corporate Research, TR ABB CH-RD 2003-26, November 2003
4. Shaw, M., DeLine, R., Zelesnik, G.: Abstractions and Implementations for Architectural Connections, In: Proceedings of the Third International Conference on Configurable Distributed Systems, 1996
5. Kostic, T., Preiss, O.: IEC 61850 UML Model, Rational Rose model file, IEC61850_Apr2003.mdl, v.5, ed. Daettwil: ABB Switzerland Ltd., Corporate Research, November 2003 (contact the authors)
6. Kostic, T., Preiss, O., Frei, C.: Towards the Formal Integration of Two Upcoming Standards: IEC 61970 and IEC 61850. In Proc. of the 2003 LESCOPE Conference, Montreal, Canada, 7-9 May, 2003, pp. 24-29
7. IEC 61970: Energy Management System Application Programming Interface (EMS-API), draft Standard, International Electrotechnical Committee, Geneva, Oct. 2002. [Online]. Available: ftp://epriapi.kemaconsulting.com/downloads
8. ISO 9506-1 and ISO 9506-2: Industrial automation systems – Manufacturing Message Specification; first edition, 2000-08-15
9. SDL Forum Society: Home page. Available at http://www.sdl-forum.org/

Experiences in Modeling for a Domain Specific Language

Steve Anonsen

Microsoft Corporation
sanonsen@microsoft.com

Abstract. Building models with a domain specific language enables targeting specific platform and framework functionality. We built a domain specific language for use in modeling applications targeting our business application framework. Such models are used for tasks including generating C# code and producing object-relational mappings for business objects. The paper briefly describes the framework and its accompanying domain specific language and then describes issues we encountered in using an unconstrained UML tool to express our models, solutions we developed to deal with those issues and observations about the suitability of UML for application to such problems. We found that making a general-purpose, extensible modeling language serve the needs of a targeted domain specific language is a lot of work and is only partially successful. We conclude that what is needed is a more general purpose framework for creating domain specific languages and tools for them.

1 Introduction

Object modeling has proven its value in application development, especially when the models are used after the design and analysis phase to implement the system and even at application runtime in a model-driven approach.

We used such an end-to-end model-driven development strategy to construct a web portal to extend two different Microsoft product lines: the Great Plains and Solomon financial management solutions. These solutions automate essential business functions such as financials (general ledger, receivables and payables), human resources, distribution (for managing inventory, ordering and purchasing) and manufacturing. They are ERP (Enterprise Resource Planning) applications for medium-sized businesses.

Given the diverse needs of businesses it is common for general applications such as those described above to be supplemented with third party software to manage, for example, the unique needs of an industry such as dry cleaning. A community of several hundred ISVs provides companion products for each of the product suites.

All of these applications and companion ISV products collectively are specific examples of applications in the *business application* domain. We created a business framework[1] for constructing new applications in this domain or for supplementing current business applications with portal and other functionality. The framework is

[1] Called the Business Portal SDK given its use for authoring web portals. However, it is not limited to portals—it is a general purpose framework for authoring business applications.

built on the Microsoft .NET Framework and its Common Language Runtime. Tools for the framework are integrated into Microsoft Visual Studio.

At their heart, business applications are on-line transaction processing applications that manage business data. In the interest of brevity this paper looks just at the business data aspects of the applications and framework, though business applications (and the features of the business framework) consist of much more, including user interface, reporting and analysis of data, customization, business logic, business process and business rules.

This paper focuses on issues in modeling applications for the business framework and solutions we provided for those issues. It starts with an overview of the domain specific modeling language (sec. 2) used to describe the structure of application data along with the UML-based visualization of the language (sec. 3). The framework implementing the semantics of this language and its supporting tools are then briefly described (sec. 4). Finally the paper sets out the problems we encountered and the solutions we developed for them (sec. 5) and concludes (sec. 6).

2 The Domain Specific Language

We developed a set of *abstractions* that are useful as the fundamental building blocks for business applications, built tools for describing applications using them and built a framework that implements their behavior. This set of abstractions and the rules concerning how they relate to each other make up a small modeling language for describing business applications that run on the business framework. It is an example of a domain specific (modeling) language, or DSL.

This business framework DSL is built to be very familiar to object modelers and those conversant with entity-relational diagrams. Some core abstractions for describing the structure of data are entity, property, identifier and association. These abstractions are elements of models for applications targeting the business framework.

An *entity* is an instance of some *entity type*, which is a type that has a durable identity by which it may be referenced. An entity type can inherit from another entity type. Examples include Customer, Order, Product and Warehouse.

An entity has one or more data members called *properties*. For example, properties on a Customer entity include its ID, name, address and credit limit.

An entity *identifier* is one or more properties of an entity that together uniquely identify an instance of that entity type. (This is analogous to a database *key*.) For example, the identifier for the Customer described above could be its ID property.

An *association* relates two[2] entities to each other and an instance of an association is a *link*. An association consists of two ends, each indicating an entity type, a multiplicity of that type, whether the association can be navigated from that end through introduction of a property, and so forth. For example, one Customer is associated to one or more Orders and it is possible to navigate from Order to Customer, but not from Customer to Orders.

A *composition* is an association that also introduces a parent/child relationship between two entities. The identity of the child need only be unique within its parent and

[2] N-ary associations were not allowed because the UML tool did not support them and they were not required for the application.

removal of the parent also removes the children. For example, an Order (the parent) has an OrderLine (the child) for each product being purchased on that order.

An *association entity* is an association that has properties and the other characteristics of an entity. It can be viewed either as an association or an entity, depending upon context, because it has all of the characteristics of both. For example, the Product-Warehouse association entity associates a Product with a Warehouse and has a Quantity property to track the quantity of a product at a given warehouse.

Attributes on each of these elements further describe them. For example, each element has a name attribute and the property element has a visibility attribute with the possible values of public, private, protected, family and family protected. It is at this point that we see how the Microsoft .NET Framework's Common Language Runtime (CLR) contributes to our DSL. Since an entity is implemented with a CLR type and an entity property is implemented with a CLR property, the DSL is shaped to fit the CLR—the ultimate runtime environment. This was an intentional choice, as it allows models used for design to seamlessly serve the application author at implementation and runtime, as described later.

The business framework DSL also has well-formedness rules that define what elements can be combined and in what ways. For example, it was not legal to create an association to a child entity in the first release of the framework.

While the above is sufficient description to support the rest of the paper, there are many other details of the DSL that are not discussed here.

3 Specifying and Visualizing Business Framework Models

A business application targeting the business framework is described with a *business framework model*—that is, a model expressed in terms of the business framework DSL described above.

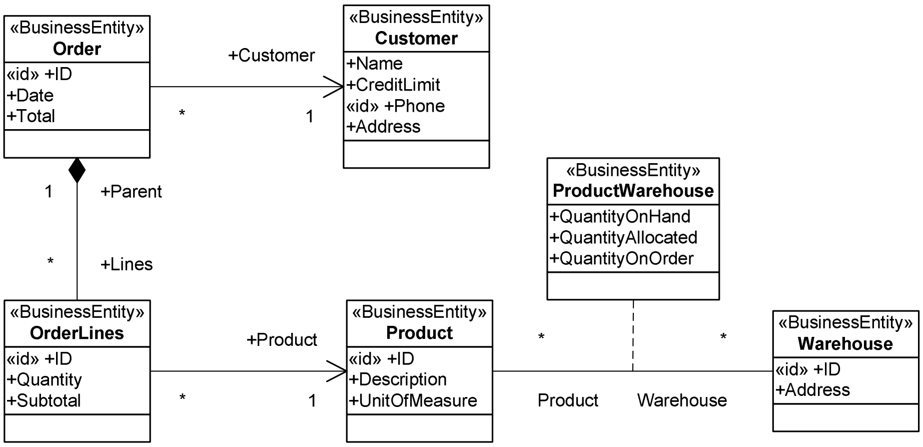


Fig. 1. Example business framework UML diagram

We developed a convention allowing the use of UML class diagrams for business framework models that we implemented in Rational XDE. Figure 1 illustrates several parts of the convention.

The DSL was expressed in UML by using the UML stereotype and tag definition extensibility mechanisms.

Table 1. Mapping from business framework DSL to UML

DSL Element	UML Visualization
Entity	Class with «BusinessEntity» stereotype
Property	Attribute
Identifier	Attribute with «id» stereotype
Association	Association
Composition	Composition
Association Entity	Association Class with «BusinessEntity» stereotype
Additional Element Info	Tagged values or, in limited cases, a stereotype (such as in the case of identifier)

At times aspects of the business framework DSL differed from UML. Properties are a case in point. In the CLR a property is effectively two methods: a getter (called when the property is read) and a setter (called when the property is written). A property is read-only if it has no setter. Further, the getter and the setter each have their own visibility[3]. Possible values are public, private, protected, family and family protected. UML attributes are different; they are read-write and the standard access modifiers are public, private, protected and (in later versions) package. (The semantics of these access modifiers are also slightly different between UML and the DSL, even where their names are the same.)

Where the DSL conflicted with UML, we either reinterpreted standard UML elements or added new tag definitions to a UML profile that was built specifically for the business framework.

A UML profile is a UML extensibility mechanism that defines additional tag definitions for standard model elements. It also defines set of stereotypes, each with a related set of tag definitions; applying the stereotype to a model element adds the related tag definitions to the model element.

In Rational XDE, applying a UML profile to a model makes those stereotypes and tag definitions available for use in that model. Rational XDE is tightly integrated into Microsoft Visual Studio, so when clicking on a model element in an XDE diagram all of the tag definitions for that element appear in the Visual Studio property sheet. Thus when a class is stereotyped, say with «BusinessEntity», the tag definitions related to that stereotype appear in the property sheet when the class is selected on a diagram.

UML profiles have their limits. They are additive only—it is not legal to in any way change the semantics of standard UML elements, stereotypes or tag definitions or

[3] This capability was not available in C# in the Microsoft .NET Framework 1.1. It is supported by the CLR and is added to C# in the beta of the next release of the framework.

to remove[4] them. That was a source of error for developers. Further, profiles do not teach modeling tools how to deal with a stereotyped element. UML profile constraints would mostly be a help for model validation rather than model construction (had XDE supported them). What is needed is a mechanism that defines how a tool should process new elements, but the UML standard does not satisfy that need.

4 The Business Framework and Tools

Application description information contained in the model is used both at design time and at runtime. *Metadata* is the term we use for this application description information.

The business framework provides a set of Visual Studio-based tools that use the metadata in business framework models to, for example, configure and direct code generation. When an entity is selected on a diagram (i.e. a class stereotyped with «BusinessEntity»), options are made available to generate code or map the class to a database with an object-relational mapping tool.

A typical approach developers use to create an entity for use in an application involves four tasks: (1) model entities and associations using a class diagram in XDE; (2) generate entity code using the code generator; (3) map entities to the database using the object-relational mapping tool; (4) and build entities and deploy. The model captures enough metadata that in many cases no coding is required—the code generator can do the work. The build and deploy task writes the metadata into a form that the business framework runtime can easily and quickly use.

The business framework runtime consists of class libraries and a set of runtime services that leverage metadata, including:

- Base classes for entities and other framework abstractions; uses entity metadata.
- An object-relational engine for reading, writing and updating entities; uses object-relational map and entity metadata.
- The ability to save an entity query by name and execute it from a web service; uses entity metadata.
- A user interface toolkit for building portal pages containing forms or the results of queries; uses entity and association metadata.
- A viewer for ad hoc query of entity data; uses entity metadata.
- A navigation service that indicates the entities associated to a given entity and allows traversing to them; uses association metadata.

This metadata is also used by runtime tools. For example, creating a portal page involves three steps that can be accomplished by an end user: (1) use the query builder to define named queries against the entities; (2) create web parts (i.e. "pagelets" or parts of a web page) to show the results from executing a named query; (3) combine web parts together into a portal page or pages.

[4] However, a tool can hide elements. A UML profile specifies the subset of the UML metamodel that it extends. A tool could hide or filter UML elements not in that subset, though the version of XDE we used did not support doing so. See also footnote 5.

Rather than using the runtime tools a developer can directly use the entities to write their business applications. They might use WinForms or ASP.NET to build their user interface or use the entities to implement a web service in the middle tier.

5 Problems Encountered and Solutions

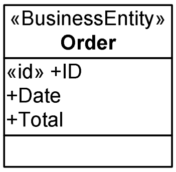
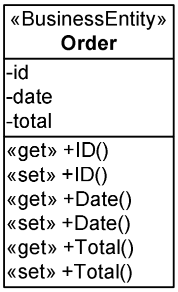
This section describes the problems we encountered, how we addressed them and some commentary about the root issues behind the problems.

5.1 Model to Code Synchronization

Initially, we attempted to use the capabilities of Rational XDE to generate code and handle synchronization between model and code. However, the version of XDE we used was too slow, did not allow sufficient control over generated code (e.g. where code was poorly formatted we could not fix it) and it reverse-engineered implementation artifacts back into the model. We ultimately solved the problem by writing our own code generator.

Two factors lay behind the difficulties in reverse-engineering from code to model. First, UML as a general purpose language does not understand features of the CLR such as properties. Second, the model and code are at two different levels of abstraction. We saw the impact of these factors when reverse engineering properties. Table 2 shows the original Order entity (left column), the code generated from that model (center column) and the entity after reverse engineering the code back to the model (right column).

Table 2. Effect of round-tripping an entity model using XDE

Model Before	Generated Code	Model After
	`class Order : BusinessEntity {` `private string id;` `private DateTime date;` `private decimal total;` `public string ID {` `get { return id; }` `set { id = value }` `}` `public DateTime Date {` `get { return date; }` `set { date = value }` `}` `public decimal Total {` `get { return total; }` `set { total = value }` `}` `// other code here...` `}`	 «BusinessEntity» Order -id -date -total «get» +ID() «set» +ID() «get» +Date() «set» +Date() «get» +Total() «set» +Total()

UML does not have the notion of a property as a get and set method. The XDE generator required changing the level of abstraction of the model to be closer to the implementation of a property.

While UML extensibility is sufficient to deal with simple extensions to its capabilities, it could not deal with this level of change. One could fault XDE here, but it is one of the more capable and extensible UML tools available. Few tools do as well as it does with this sort of problem. Ultimately UML extensibility is not rich enough to change the behavior of tools in any but the most simple of cases.

We did not solve the round-trip engineering problem. Our tools only did forward code generation. This was less problematic than may seem, however, because so little code was required to implement the system—most of it was declarative in the model. As described in the conclusions, we are pursuing a new direction on round-trip engineering that removes the synchronization problem rather than solving it.

5.2 General Purpose Nature of UML

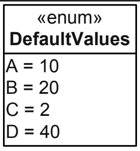
UML is a general purpose language that does not have precise semantics in a particular runtime environment, as illustrated in the discussion on properties in the previous section. Using UML to solve a particular problem on a particular platform requires mapping its concepts to the concepts of the platform and using its extensibility mechanisms to fill holes.

We created an extensive modeling style guide that described such a mapping from UML to the business framework DSL and the .NET Framework's Common Language Runtime. It described the semantics of each model element in terms of the CLR and described modeling conventions for modeling DSL concepts and CLR concepts not present in UML, such as events, properties, enumerations, delegates and (surprisingly) constructors. For example, the following is an excerpt from the guidelines for modeling a CLR enumeration type:

Enumerations

- Model an enumeration as a class stereotyped with <<enum>>
- Suppress the operations (methods) section since enumerations cannot have methods
- Make all attributes public since all members of an enumeration are implicitly public
- System.Int32 is the default underlying type for an enum. To indicate use of a different underlying type, derive from that numeric type.

Enumeration	C# Code

 «enum»
DefaultValues
A = 10
B = 20
C = 2
D = 40
	`// type is integer in this example`
	`public enum DefaultValues {`
	`    A=10,`
	`    B=20,`
	`    C=2,`
	`    D=40`
	`}`

The modeling style guide also served as the starting point for a requirements specification for the XDE code generator and later the code generator we wrote.

Our goal was to make the modeling language fit the expectations of those who knew UML, and we succeeded in making the conventions familiar to moderately experienced UML users. What surprised us was how much of the UML language even experienced UML users do not know. (For example, is a multiplicity of "*" synonymous with "1..*" or "0..*"?)

Mapping UML to the business framework DSL and CLR uncovered a host of ambiguities. (That was less surprising, given UML's general purpose nature.) Designing the mapping was a time consuming affair that involved some of the more senior people on the team.

We only built tools and conventions for the static structure diagram, likely the best known and most used part of the language. One of the things we did not do was describe what parts of the UML were not meaningful to the business framework. This contributed to the problems described in the next section.

5.3 Malformed Models

Commonly developers produced models that had mistakes in them. This resulted in errors when generating code, mapping entities or at runtime when a malformed entity was used. Our solution was to create a model validator to ensure models were correct before other tools were used. The model validator moved error discovery earlier in the process, increasing productivity and reducing developer frustration.

However, the model validator was a reactive solution. If we had more control over the Rational XDE design surface many issues could have been avoided simply by disallowing certain kinds of models. Consider some of the errors developers made:

- Forgot to stereotype entities and other model elements.
- Did not mark any entity properties as its identifier.
- Created an association to a child entity (such as to the OrderLine in the above diagram); doing so was not supported in that release of the framework.
- Used multiple inheritance; this can be hard to detect, since the two generalization relationships can be created on two diagrams.
- Used characters that are illegal in CLR identifier names.
- Did not set business framework tagged values properly.
- Used UML features that had no meaning to the business framework DSL.

So we found ourselves wanting additional extensibility features in XDE. However, XDE is a UML tool and some of the issues were differences between UML and the business framework DSL. As noted above, UML profiles cannot in any way redefine standard UML elements.

UML has features the business framework DSL does not. For example, UML supports multiple inheritance. However, the business framework DSL has features UML does not. These differences are the heart of the problem.

The differences also created problems for application developers as they did not always know what parts of the UML were valid for them. The tool itself should have indicated what was usable and what was not. By design the UML does not supply guidance for domain specific constructs either, an important capability when modeling for a DSL.

6 Conclusions

We drew three conclusions from our experiences.

Model-Driven Development Is Effective
Model-driven development proved effective for the business application problem space. In model-driven development the application developer describes much of their application rather than writing code to implement it. Coding is still a part of the process, but relatively speaking imperative code is a smaller part of the overall application than the metadata describing the application.

The majority of the two portal projects described in the introduction were built without imperative code. Developers indicated the behavior they needed with tagged values and the tools and runtime did the work. They also used the same tools provided to customers for much of the development work, again without coding.

Model to Code Synchronization Should be Avoided, Not Improved
We came to the conclusion that the core issues in model synchronization are the related issues that (1) model information is present in more than one spot and (2) developer code is intermixed with system generated code. The latter problem is simply a manifestation of the former.

Going forward we are designing the tools and frameworks to avoid redundant model information. In particular, most model information will not be visible in the code. One powerful mechanism Microsoft is using for this is the partial class. Consider the following C# code sample

```
// in framework_order.cs: framework class
public partial class Order : Entity { // generated
   // framework generated code
}
// in order.cs: developer class
public partial class Order {
   // developer written code
}
```

The C# compiler takes the code from these two files and integrates them together into one class at compile time. Thus the framework can regenerate its class at any time without affecting developer code. The same capability is available in VB.NET. Describing this in detail will be left for another paper.

UML Is Not Suited as a Basis for a DSL
We came to question the suitability of using UML tools as the basis for building tools for domain specific languages. Used informally and semi-formally UML is incredibly helpful for team communication and design and we expect it to continue to play that role for us going forward—UML is used regularly in our own designs today.

However, our scenario was to build a design tool for a DSL that describes the vocabulary of a specific framework hosted on a specific runtime. Such tools require crisp and detailed semantics to be usable, so as a result we tested the limits of the UML metamodel and extensibility mechanisms. As it happens the UML explicitly is not intended to solve this sort of problem, for two reasons.

First, UML lacks a well-defined mapping to any given framework and execution environment. Stated differently, the UML is a general purpose language, not a DSL. If the DSL is not completely compatible with the UML then changes in semantics are needed. UML provides such extension capabilities through profiles.

Second, UML profiles are purely additive[5]. But a DSL by its very nature is likely to change the design vocabulary to the extent that it strains or exceeds the capabilities of the UML extensibility mechanisms. The UML specification indicates that true semantic changes to its metamodel involve creating your own metamodel using MOF[6] rather than using UML.

Developer tool usability is enhanced by working directly with DSL concepts rather than requiring mental mapping from awkward UML expressions of DSL concepts to their DSL equivalents. Such mapping is avoided only if the DSL is a superset or subset of UML, which was not true of our DSL and is not true for many DSLs.

Further, UML extensibility was specifically not designed for this kind of tools extensibility. While allowing constraints in a profile, it does not dictate the constraints language[7] or provide any other mechanisms for directing a tool in processing a stereotyped element such as giving the developer interactive guidance when modeling.

Adapting a UML tool to be the modeling tool for a framework DSL requires a significant amount of work:

- Map from UML to the DSL and the execution environment.
- Define a profile or profiles to handle the new features of the DSL.
- Develop code generators and other tools to implement the model.
- Implement add-ins for the selected UML tool to integrate custom tools with it.

Once done, you are still left with unneeded UML features or semantic differences that UML extensibility (or at least the XDE implementation of it) could not erase.

Our conclusion is that UML simply is not intended to support a DSL and framework. We found that shoehorning a general-purpose, extensible modeling language into serving the needs of a targeted DSL is a lot of work and has mixed results.

We believe a more fruitful approach, and one we are now pursuing, is to formalize the metamodel for the DSL and create a custom visualization for it (consistent with the principles described in [2]). Diagrams then will have just features belonging to the DSL and can be tuned to fit its vocabulary. In the case of the business framework DSL the diagrams will look very much like UML class diagrams. But the reality is that a the semantics of a given DSL need not and often will not match UML, even if the notation used on diagrams is very similar or the same.

To support this approach, we see the need for (and we are producing) a more general purpose framework for creating domain specific languages and tools for them.

[5] "A fundamental constraint…is that extensions must be strictly additive to the standard UML semantics. This means that such extensions must not conflict with or contradict the standard semantics."[1] However, a tool may hide those UML elements that are not extended by or referenced in the profile. (See "applicableSubset" in [1] and "metaclassReference" in [4].)

[6] "Profiles are…the 'lightweight' built-in extension mechanisms of UML, in contrast with the 'heavyweight' extensibility mechanism as defined by the MOF specification. …there are restrictions…to ensure that any extensions defined by a UML profile are purely additive. Such restrictions do not apply in the MOF context where…any metamodel can be defined."[1]

[7] However, UML does provide OCL and uses it extensively in its own definition.

References

[1] Extension Mechanisms Overview, §2.6.1. OMG Unified Modeling Language Specification Version 1.5. Object Management Group, March 2003. http://www.omg.org/cgi-bin/doc?formal/03-03-01.

[2] J. Greenfield and K. Short. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley and Sons, 2004. ISBN 0471202843.

[3] James Rumbaugh, Ivar Jacobson and Grady Booch. The Unified Modeling Language Reference Manual. Addison Wesley, 1999.

[4] Profiles, §18.13.5. UML 2.0 Superstructure Specification. Object Management Group, August 2003.

[5] http://www.omg.org/cgi-bin/doc?ptc/2003-08-02.

Six Lessons Learned Using MDA

Stephen J. Mellor[1] and Leon Starr[2]

[1] Accelerated Technology, 739 N. University Blvd., Mobile, AL 36608 U.S.A
Stephen_Mellor@mentor.com
[2] Model Integration, LLC, 500 Botany Court, Foster City, CA 94404 U.S.A
leon_starr@modelint.com

Abstract. The principles behind MDA have been applied over several decades with varying degrees of automation and support. Changing technology and standards render many "lessons learned" dependent to a considerable degree on implementation specifics that will become less relevant as the standards and technology grow and mature, but the process behind MDA has changed somewhat less and years of application allow some broad conclusions to be drawn about how best to put MDA into practice. We intend to supply here some hard-learned lessons that can be applied immediately on an MDA project.

1 Background

At one level, model-driven architecture (MDA) is a set of standards promulgated by the Object Management Group (OMG), which at present is insufficient to build complete systems. Yet MDA is also the result of years of application of technologies centered on the notion that "design" can be captured as a set of mappings, or automated transformation rules from one language to another.

The authors have both been defining and applying that notion for over twenty years. In so doing, we have seen many project successes and many failures too. This paper describes key lessons learned from those experiences, which we support with anecdotes drawn from real-time and embedded projects, though we feel confident the lessons apply to other types of system too.

Throughout, we assume executable models of some sort. When we say "model," we usually mean an executable model of the application. We also use the term "architecture" to mean the abstract organization of implementation technologies and their use as realized by a model compiler—a compiler that compiles application models normalizing them to a single infrastructure defined by the architecture.

We conclude by making some larger observations about the place of modeling and MDA.

1.1 Lesson 1: Maintain and Establish a Domain Chart

MDA relies on the construction of models, with their respective metamodels, and mappings between them. We have learned that it is critical both to establish what the models and mappings will be early on, and to revisit these choices throughout the project. Models capture selected subject matters called domains. Bridges exist as systems of mappings and marks between selected domains. This content can be represented on a *domain chart*, a non-UML-standard sketch that depicts domains and bridges.

The key aspect of a domain chart (more accurately, the thinking behind the construction of a domain chart) is the establishment of assumptions about what will be modeled. An example is a model of a bank, which under many interpretations includes authentication and authorization. But, it is also possible to abstract out security as a separate domain with its own model that is then reusable in other contexts.
Making this choice of abstraction clear to the team is critical to avoid endless rehashing of the models. Since security is only one possible issue (to what extent should values from the user interface be pre-validated? to what extent should credit ratings be factored out? and so on), all too often we have seen modeling teams flounder around making arbitrary assumptions about what should be modeled here and what elsewhere. The solution is to make these assumptions explicit.

Even a bad domain chart is better than committing to one colossal tangle of modeling. Modelers and managers must be on the look out for any discussion about what needs to be modeled where. They should immediately refer to, and possibly update, the domain chart. Often, the issue seems to be a local one (Should we authenticate the customer when a deposit is made?), but often it is more global: Should we factor out security as a separate model? It is all too easy to address the local issue and fail to recognize the global opportunity for reuse.

We note in passing that UML needs a domain chart to encourage an MDA process. Presently, we misuse the package diagram, using packages to represent models and dependencies to capture mappings between those models. Of course, "dependency" is precisely what we do not mean, as each model is orthogonal from the others, and mappings are explicitly not expressed as dependencies in MDA.

1.2 Lesson 2: Force Limitation of Requirements

A vague statement like "model the Navy," guarantees that the modeling work will never finish. Equally deadly is the all too common statement "model version 1.0, but leave hooks in to support version 9.0," which in practice is often functionally equivalent to "model version 9.0 in the absence of any idea as to what that might be."

There are two kinds of scope creep. "Horizontal" scope creep is exemplified by modeling a too-large system, such as the entire Navy or version 9.0. The result of this type is usually late delivery and lack of direction caused by lack of feedback from a delivered system. The second type is "vertical" scope creep caused by generalizing a new service domain, such as security in the example above, with insufficient information. Premature generalization can lead to the construction of service domain models that provide more functionality than required for the particular client, and often fail to deliver the small amount of functionality that is required. Both kinds of scope creep are bad.

This means there must be some mechanism for limiting the requirements and enforce the limits when the modeling team begins to wander. The key word here is "mechanism." Ad hoc excision of superfluous requirements requires constant vigilance from management and project leads, which is an undue burden on management and can lead to low morale when all project staff hears is the word "no." There are several approaches to this problem, including constant interaction with customers, priority setting by sequence, time boxing and so on. Whichever mechanisms you choose, it must become a part of the culture to build only what is required *now*.

This lesson is worth repeating and reinforcing because all too often models are perceived as superfluous in and of themselves, not relevant to the all-important problem of generating systems.

1.3 Lesson 3: Charter an Architecture Team Early

An architecture team must be chartered to create an implementation plan soon after model development begins. Without this plan, modelers may not see a direct path from models to code and may eventually lose confidence in the approach. With an active implementation plan in place, however, effort will be focused on coding related problems rather than wringing hands in unconstructive anxiety. As compilation capabilities become available, models can be executed and tested thus increasing confidence that the approach will work. This puts pressure on the application modeling team to produce converging models. The models become real, executable entities rather than abstract concepts. This proves progress and is the ultimate confidence builder.

In one project, extensive requirements mandated the construction of a large model set, which would have taken at least two years to complete with the available team. Not perceiving a tangible path to implementation, the team became confused as to how various design-level details would be addressed, and began to lose confidence in the approach during the first year. This created uncertainty as to the level of detail required to complete any given set of models, which led to eventual project failure.

1.4 Lesson 4: Build—or Buy—the Architecture in Parallel

The simplest way to address the lesson above is to buy a model compiler. By compiling newly constructed application models, you can also verify if the performance properties meet the requirements. If they do not, then you should begin an effort to determine the abstract organization of the software and its expression in a model compiler in parallel with the application modeling effort.

Application modeling and architecture development need not be carried out in sequence since you can build the model compiler without concern for the semantics of the application. You can have one team modeling an application in one room, and down the hall you can have an architecture team figuring out how to implement persistent classes efficiently, so long as you do have a common knowledge of the structure of the modeling language.

To ensure that the model compiler is "efficient" enough, you need to compile the application models as they are built, and check their performance against requirements. Consequently, two teams may operate in large measure independently, meeting on occasion to ensure that the model compiler will properly translate the models, and that the model compiler is meeting performance requirements.

1.5 Lesson 5: Order the Work Depth First

A fully breadth-first approach to modeling is risky. It can take longer to model a given set of requirements than you think, especially with a team new to modeling. Instead, the team should first model the core features in each modeled domain and then drive toward an engineering version of the whole system. This early implementation may not produce a deliverable product, but it will allow the team to evaluate the modeling and implementation process early on. To do this, however, the initial set of requirements must be whittled down to the bare bones. Multiple cycles through the translation and test phase may be necessary before a deliverable product system version is available.

On the other hand, a factor contributing to disaster is premature modeling of service domains. In one project, an initially straightforward parameter management service domain became the misdirected focus of much effort. But complex interactions and dependencies among parameters were application specific and could not readily be addressed at such a generic level, certainly not without a clearer understanding of the application. Consequently, we learned that it is preferable to model a significant portion of the application before modeling in depth domains that provide services to the application. This is because application details will define and constrain requirements on otherwise runaway abstract and generic support services.

The conclusion is that the work needs to be sequenced depth first down the domain, building only enough of each service domain to make an executable system. As understanding grows, service domains can be re-factored, and the domain chart updated. The least risky approach is to order the work depth first without attempting to build complete service domains the first time. A variation of this theme is Lesson 4 above. Rather than attempting to build the complete, correct model compiler from first principle, use one that's available and iterate from it. Often, less iteration is required than you first think.

The architecture can be an exception to this rule about service domains. While the architecture domain is in fact a service domain, a significant subset of its requirements may be clear prior to application development. If the chosen modeling language for multiple client domains is rigorously defined, precise and executable, then the requirements for model execution are already embodied in the language definition. Furthermore, there are typically many engineers well versed in the characteristics of the likely implementation technologies for the given project. Consequently, progress on the architecture domain, as stated in Lesson 4 may proceed immediately. Other service domains, however, should follow the plan set here in Lesson 5 since they will likely benefit from the requirements clarification resulting from sorting out the application domain first.

1.6 Lesson 6: Manage Process Change

Model-driven code generation imposes a paradigm shift on the management, engineering and technical staff. It is the process rather than the development tools that impose this fundamental change. The engineering staff will use new cognitive and engineering processes that require time for adjustment and assimilation. Day-to-day management tasks also change.

In one project, automated code generation was the most significant achievement. By compiling each domain separately, using a version control management system and scripts for overnight batch processing, the team could build and execute nightly jobs for each developer. A tester could then run through an ever-increasing number of automated regression tests. The next morning each developer and tester received a detailed, diagnostic report on the submitted job from the previous night. The number and rigor of the regression tests increased over time. By this stage the team had a fully tested, executable product that could be shipped on a day's notice.

2 Observations

This paper began as a set of project histories, each laid bare to expose the elements of success and failure. Each project history finished with a lessons learned section, which led to some undeniable similarities, but also some differences. Drilling down further, we noted that some lessons learned did not repeat because they had in fact been learned and put into practice. We re-factored the paper to focus on lessons learned.

Summarized the lessons learned in this manner, we observed that many of them relate strongly to the Extreme or Agile movement. These movements emphasize construction of complete executable systems as soon as possible. The lessons learned primarily focus on rapid progression from concept to implementation, rapidly filling in the stepping stones. Indeed, our most striking example of the need to manage process change was the construction of the daily build tools motivated by the rapid and visible integration of models.

Perhaps then we draw these conclusions because of our emphasis on executable models. Certainly executable models offer many advantages claimed for code by the Extreme/Agile movements, but these lessons were learned in the context of making modeling work in an impatient modeling environment—impatient not only for the usual reasons of customer delivery, but also because of the need for modelers new to the game to see encouraging and meaningful results.

Our experience indicates that one of the key impediments to MDA adoption is the *a priori* assumption that model driven code cannot possibly work. Even if it does work on some projects, it is too risky for *my* project. Building a deliverable system early allays these fears. We hope that our lessons learned will help provide a pragmatic and successful approach to MDA adoption.

Applying MDA and UML in the Development of a Healthcare System

Chris Raistrick

Kennedy Carter Limited,
Hatchlands, East Clandon, Guildford, Surrey, GU4 7SJ, UK
chris.raistrick@kc.com

Abstract. The UK National Health Service (NHS) is supported by an established and diverse set of systems. These systems are being enhanced to support widespread storage and distribution of electronic patient records. Naturally, the security and privacy of patient data is paramount, and a sophisticated set of access control and registry capabilities is required to ensure that only authorised persons have access to patient data. This describes how MDA and UML were used to:

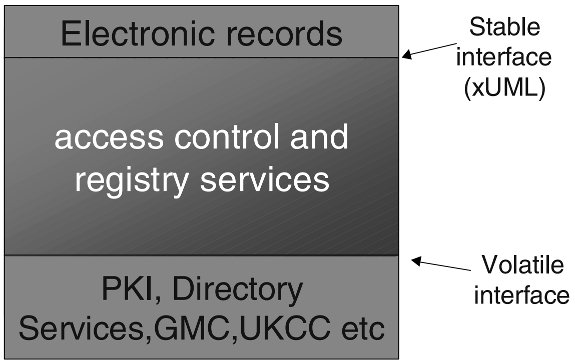
- Model the new access control capabilities in the form of an executable UML model, allowing rapid stakeholder feedback and resulting in a tested, fully executable specification to form the basis of a procurement contract;
- Specify the capabilities of existing key components, and proposed bought-in components, to facilitate system integration;
- Enable automated code generation onto the wide variety of platforms in use within the NHS.

1 Introduction

The NHS Information Authority (NHSIA) provides a range of patient information services to the NHS. This paper outlines how MDA and UML were used in the context of an extension of the processing of clinical data to provide a "patient-based electronic record".

The project set out to design and pilot parts of the infrastructure to allow demonstration of compliance with both the national confidentiality policy and the cryptography strategy in respecting patient confidentiality, including consent to the sharing of information.

The project scope included the specification of software to provide interim access control, consent_management and user registration services in a number of NHS care providers. Requirements drawn from relevant legislation and standards documents and from subject matter experts within the Authority were elicited and incorporated in the UML specification. Because of the volatility of the requirements in the areas of consent and access control, it is important that only one specification, in the form of a platform-independent model, be maintained, and that multiple platform-specific implementations are automatically generated from it. The systems with which the software must integrate are based upon a diverse set of technologies and architectures, and therefore the specification must be expressed in a form that is both independent of, and facilitates mappings to, these different platforms.


Electronic records
access control and registry services
PKI, Directory Services,GMC,UKCC etc
Stable interface (xUML)
Volatile interface

The challenge is to ensure that a number of initiatives are brought together in a consistent and coherent way, within the focus of 'Access Control and Registry Services', to be of direct and practical use to those developing Electronic Health Record and Electronic Patient Record Systems.

The project had two main objectives.

1. To provide a stable application interface between Electronic Record Systems and Access Control and Registry Systems independent of changing technologies and future national procurement decisions. These systems include those provided as part of Public Key Infrastructure (PKI) Services, online directory services and professional registration services (e.g. those provided by the General Medical Council (GMC) and the United Kingdom Central Council for Nursing, Midwifery and Health Visiting (UKCC).
2. To support organisational use of access control and registry services, allowing a variety of users to have access to healthcare records, while respecting patient confidentiality, including consent to the sharing of information.

To achieve these objectives the project set about designing and building prototype registry and access control systems to be integrated with target EHR and EPR systems being developed concurrently. The aim was to demonstrate how compliance with national confidentiality policy and the cryptography strategy is to be achieved and will entail an exploration of the interfaces needed between a public key infrastructure, related access control services, sources of user and patient identity, and electronic record/patient information services.

1.1 Project Scope, Exclusions, and Interfaces

The specific services to be provided through the stable interface to Access Control and Registry Services include:

- Consent management;
- Registration management and identification of organisations, and healthcare professionals including their addresses, access mechanisms (addresses and public keys), roles and geographic locations;
- Certification and authentication of those requiring access to patient records;
- Access management (ensuring that those who need access actually get it), and access logging.

1.2 Requirements Driven Modelling, with Model Driven Architecture

The project coined the phrase "Requirements Driven Modelling, with Model Driven Architecture" [1], to reflect the adopted approach. It was regarded as essential to establish and maintain clear links between the requirements and the emerging models. The project set out to capitalise on prior work, such as scenario definition, and comply with pre-existing standards, such as healthcare system architecture definitions. The basic steps followed were:

1. **Requirements:** Establish the domains of interest, and the requirements within those domains. The resulting Domain Model shows the domains (or components) that will make up a typical system. It shows the dependencies between the existing components, the third party components (e.g. certification products), and the newly developed access control components. The domain chart is a derivation of the "*Domain model for National Confidentiality Infrastructure*" held in the document "*A Portal for Controlled Access to Patient Information Services*".
2. **Modelling:** Use UML, the Government standard for representing software requirements, to build an executable specification, or Platform-Independent Model (PIM) for each of the new domains to be developed. This allowed developers to highlight key issues early, particularly with regard to the technical integration with pre-existing systems. Each domain PIM specifies the set of services provided by, and required by each domain. Each service, or operation, is defined in terms of its name, its parameters, contractual semantics (open/closed, blocking/non-blocking), and a textual description of its purpose. These are used as the basis for integrating domains to form a complete system.
3. **Demonstration:** Use the PIM to generate automatically a demonstrator for the relevant stakeholders. This included the use of an existing library of scenarios, under development within theNHS, addressing the roles and responsibilities of the various actors. The scenarios were captured using executable UML for use as data to drive the demonstrator;
4. **Procurement:** Use the PIM to develop indicative costs for the development, implementation and operation of all aspects of the system by service/solution providers.

The requirements, modelling and demonstration process was executed in a modular and iterative fashion. Relatively self-contained domains (see the Domain Model) were identified and their interfaces specified. Each domain was then further developed and tested in isolation prior to being integrated with other components to form the demonstrator.

The following sections describe in more detail each of these phases.

2 Requirements and Domains

2.1 Requirement Driven Modelling

The requirements for access control were sourced from a wide variety of documents and stakeholders, and it was vital that each requirement could be traced into the UML models. To this end, the iUML modelling tool, used by the project to build and test

UML models, was configured using the requirements schema in Fig 1 below, represented as a UML class diagram.

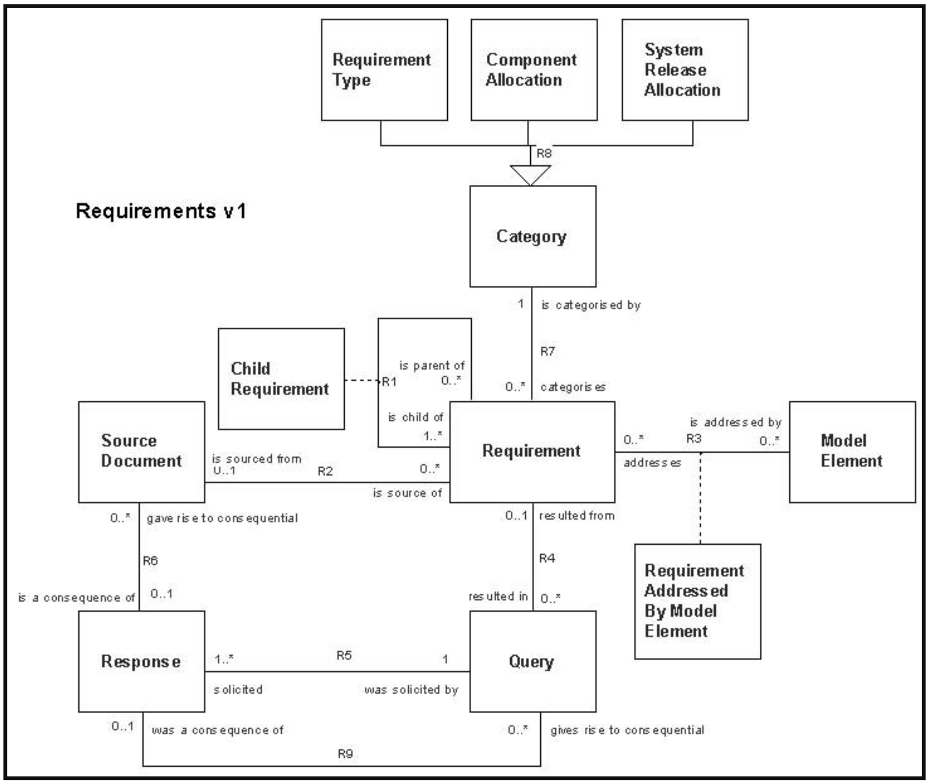


Fig. 1. Requirements Schema

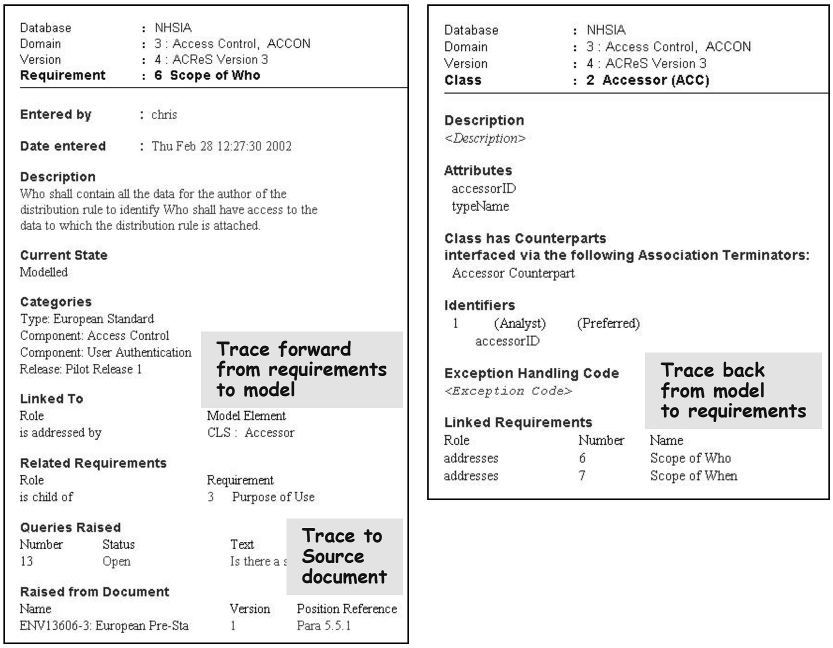
This allowed requirements to be categorised, arranged into parent-child hierarchies, traced back to their source, or forward into the models, as illustrated in Fig 2 below.

To deal with requirement volatility, this scheme also captures the "audit trail" of query-response-modify requirement-modify model, providing the historical record so important for understanding why the UML models look the way they do.

Each requirement also went through a set of lifecycle phases, as shown in the statechart diagram in Fig 3 below, which is represented using the Moore formalism as described in [3] and [5].

2.2 The Components (Domains)

It is helpful to think of an executable UML model as an attempt by an analyst to formalise precisely the requirements expressed in ambiguous language by various subject experts and stakeholders. Although the analyst may not initially understand all



Database	: NHSIA
Domain	: 3 : Access Control, ACCON
Version	: 4 : ACReS Version 3
Requirement	: **6 Scope of Who**

Entered by : chris

Date entered : Thu Feb 28 12:27:30 2002

Description
Who shall contain all the data for the author of the distribution rule to identify Who shall have access to the data to which the distribution rule is attached.

Current State
Modelled

Categories
Type: European Standard
Component: Access Control
Component: User Authentication
Release: Pilot Release 1

Linked To

Role	Model Element
is addressed by	CLS : Accessor

Related Requirements

Role	Requirement
is child of	3 Purpose of Use

Queries Raised

Number	Status	Text
13	Open	Is there a s

Raised from Document

Name	Version	Position Reference
ENV13606-3: European Pre-Sta	1	Para 5.5.1

Database	: NHSIA
Domain	: 3 : Access Control, ACCON
Version	: 4 : ACReS Version 3
Class	: **2 Accessor (ACC)**

Description
<Description>

Attributes
accessorID
typeName

Class has Counterparts interfaced via the following Association Terminators:
Accessor Counterpart

Identifiers
1 (Analyst) (Preferred)
accessorID

Exception Handling Code
<Exception Code>

Linked Requirements

Role	Number	Name
addresses	6	Scope of Who
addresses	7	Scope of When

Trace forward from requirements to model
Trace to Source document
Trace back from model to requirements

Fig. 2. Requirement Tracing

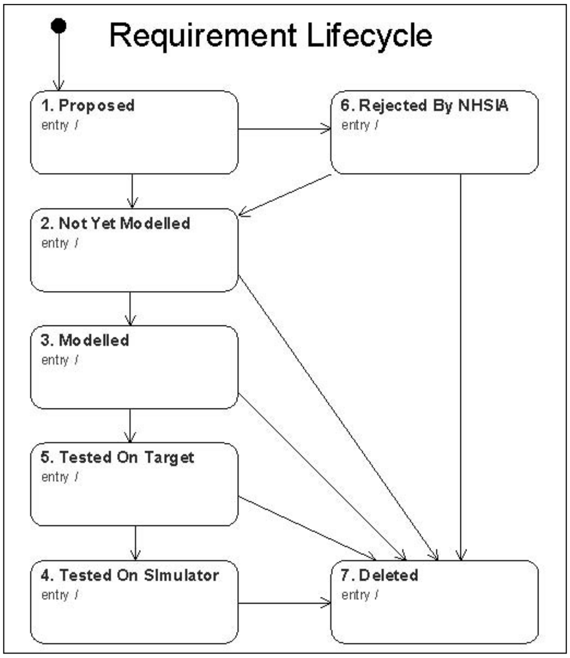


Fig. 3. Requirement state machine

the concepts, his questions and interaction with subject experts always expose hidden assumptions, elicitation of missing information and general clarification. This is one of the benefits of a model-driven approach to specification.

To formalise the conceptual content of the requirements and the behaviour they define; models were designed to lend themselves to ongoing modification. This was achieved by a combination of generic modelling, and isolation of volatile issues in separate domains, which nonetheless offer a stable interface to the rest of the system. More on this later.

In order to be able to capture and control the complexity of the overall system, it was necessary to define several inter-related domains. By minimising domain interdependence we ensure that that each domain can be defined and tested independently.

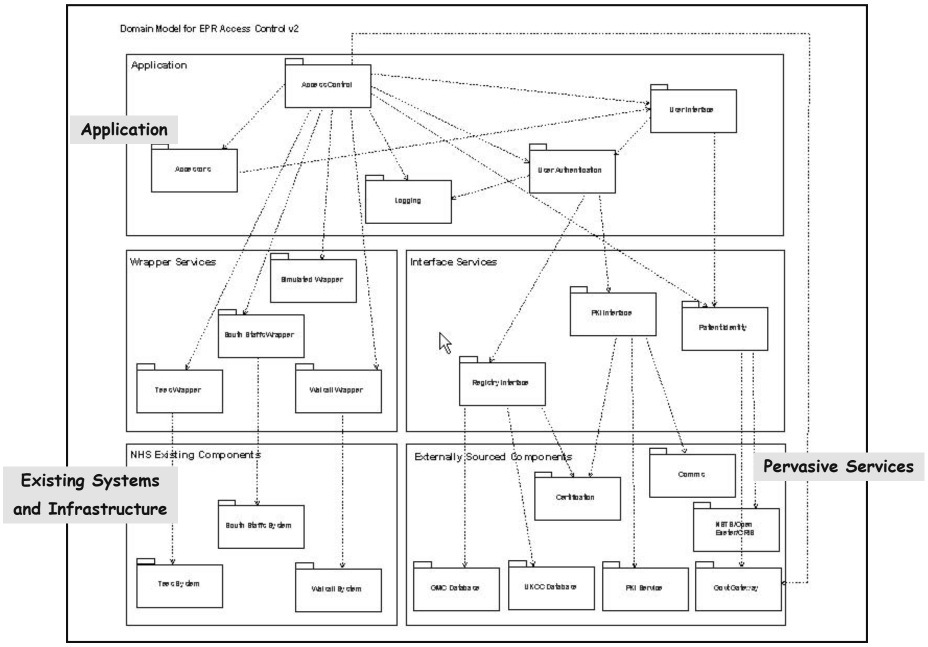


Fig. 4. Domain Model

Although domains publish contracts for required and provided services, there is no requirement to reconcile these contracts until the domains are integrated to form a system under test. Thus, completely decoupled domain models were the norm, and each one was developed and tested independently of the others, subject of course, to the satisfaction of the requirements allocated to them. The work involved in integrating these domains was minimized by building a set of domain-level sequence diagrams, based upon a set of use cases. These sequence diagrams gave an early indication of the services required by and provided by each domain in the system, and provided an agenda for defining the interface offered by each domain.

This domain model diagram in Fig 4 shows the various components (domains) comprising the proposed solution.

The domains were grouped to aid readability, and provide a clear separation between new components (in the "Application" layer), and the existing and bought-in components to with which the new components must integrate. "Wrapper" and "Interface" services were included to allow definition of a standard set of interfaces for accessing patient data, registry services and encryption capabilities. This was intended to make integration with existing components, and COTS components more straightforward.

2.3 Modelling the New Access Control Requirements

The majority of the new requirements were modelled in the "Access Control" domain. To support this domain, and allow construction of a demonstrator, a "Patient Records" domain was also built, conforming to the emerging standard architecture for electronic patient records. The "Aaccess Control" domain is a client of the "Patient

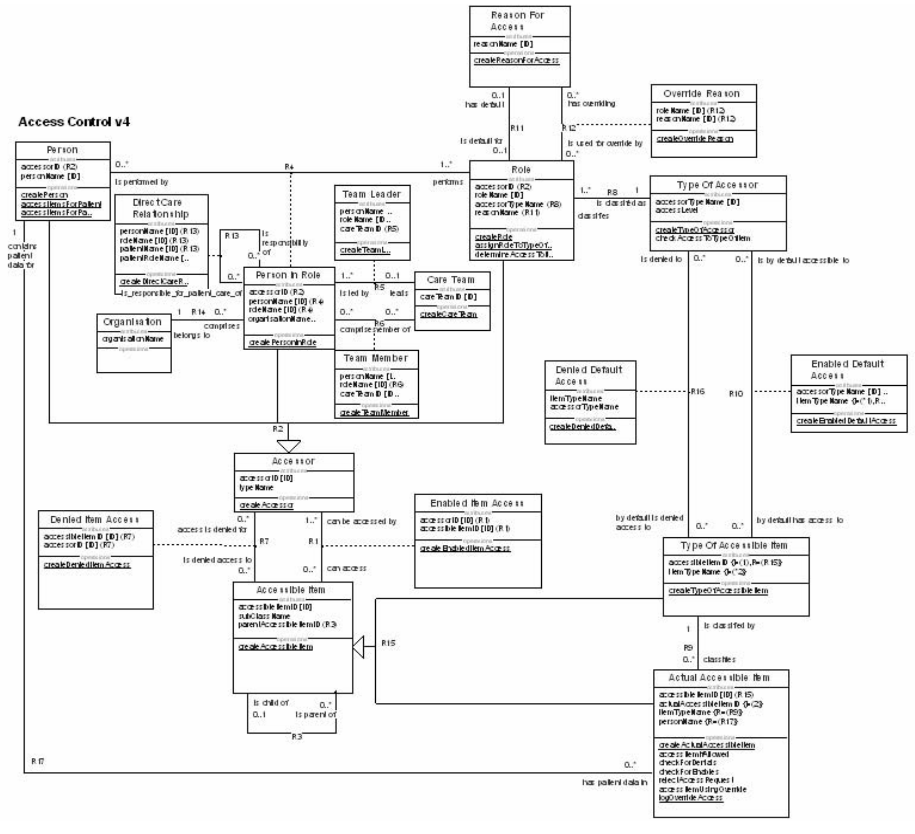


Fig. 5. Access Control domain class diagram

Records" domain (and any other legacy domains containing patient data, to which the new access control capabilities are to be added). The UML class diagrams for these are shown in Fig 5 and Fig 6 below. It is beyond the remit of this paper to delve into the detail of these domains, but if the class diagrams are well constructed, they should convey the essence of the requirements without too much elaboration. One key point to note is that the "Accessible Item" class in the "Access Control" domain is linked `(via a "counterpart association" – see later) to the "Architectural Component" class in the "Patient Records" domain. This approach allows the "Access Control" domain

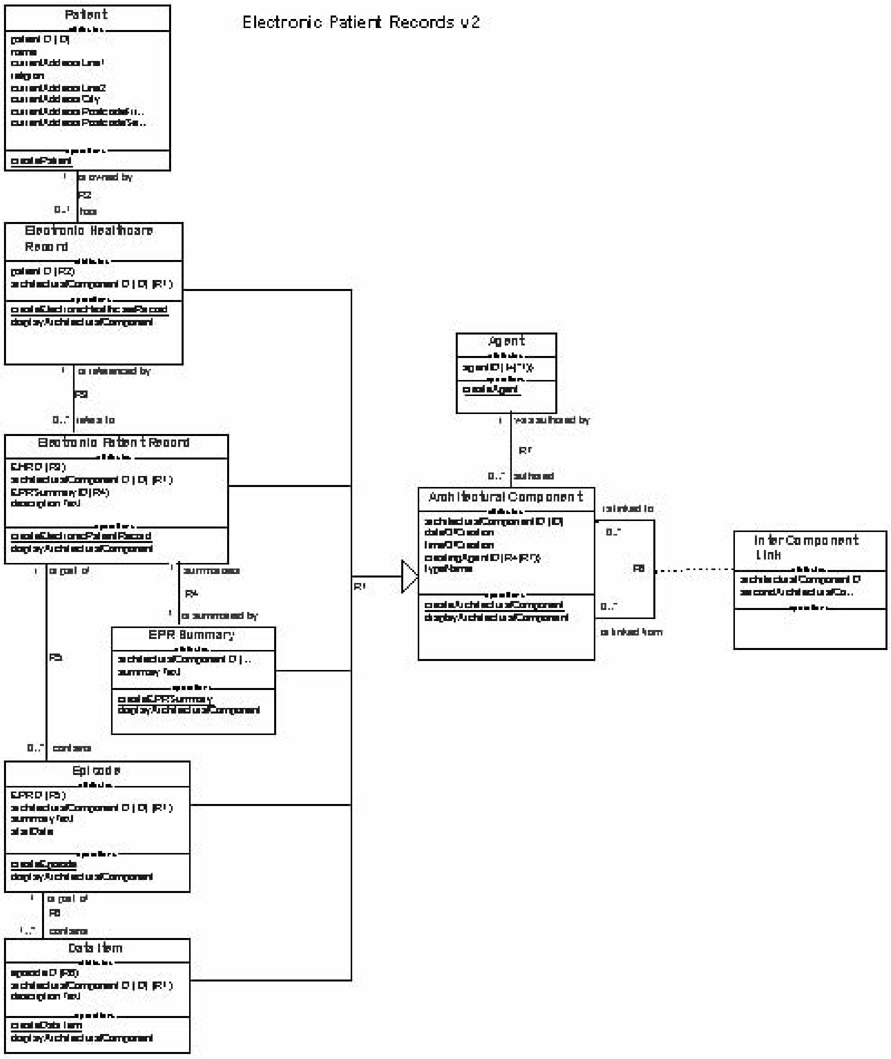


Fig. 6. Patient Record domain class diagram

to focus purely on controlling access to data items, without knowing anything about the structure or content of those items. This is vital, given the range of legacy patient record systems, and record structures with which this domain must interact.

The services provided by this domain enable the setting of permissions by all concerned according to their respective roles and responsibilities and that only those duly authorised will gain access. It allows health care roles to be defined for this purpose for use as defaults throughout the NHS. Such general defaults are subject to modification according to specific roles filled by an individual within an organisation.

The services provided can act as an exemplar for the NHS as a whole, and are scalable to satisfy emerging requirements.

2.4 Modelling the Standard Patient Record Architecture

This domain was built to support testing and demonstration of the Access Control component. It was populated with realistic data, extracted from standard scenarios, to allow meaningful stakeholder demonstrations.

Notice that the classes representing data to be protected are modeled as subclasses of "Architectural Component". This means that the client "Access Control" domain can invoke the polymorphic operation "displayArchitecturalComponent" operation on "Architectural Component", and can remain unaware of the structure and content of the patient data itself.

3 Maintaining Patient Record Architecture Independence

Although there is an emerging standard "architecture" for Patient data, there are of course many existing systems, each of which holds patient records in a particular way. It was therefore necessary to establish and maintain a clear separation between the patient data to be protected and the protection rules that are to be applied to those data. This was achieved by modelling the Patient Data in the "Patient Data" domain, and the access control rules in a separate "Access Control" domain, in keeping with the data-driven modelling techniques described in [2], [4] and [5].

Each of these domains has a class that represents an item of patient data. In the case of the Access Control domain it is the "Actual Accessible Item" class, which has associations (via the "Accessible Item" superclass) to classes that capture the access policies, such as "Enabled Item Access" and "Denied Item Access" as shown in the model fragment in Fig 7 below:

In the case of the Patient Records domain, there is an "Architectural Component" class, with subclasses corresponding to the various types of patient record, as shown in the model fragment in Fig 8 below:

The "Access Control" domain is a generic component, to be used to protect data held in many different forms across a number of patient record systems. Therefore, the "Access Control" domain must be unaware of the architecture of the data it is protecting. The required mapping between "Accessible Item" and "Architectural Component" is achieved using a "counterpart relationship", an association that spans classes in different domains, as illustrated in Fig 9.

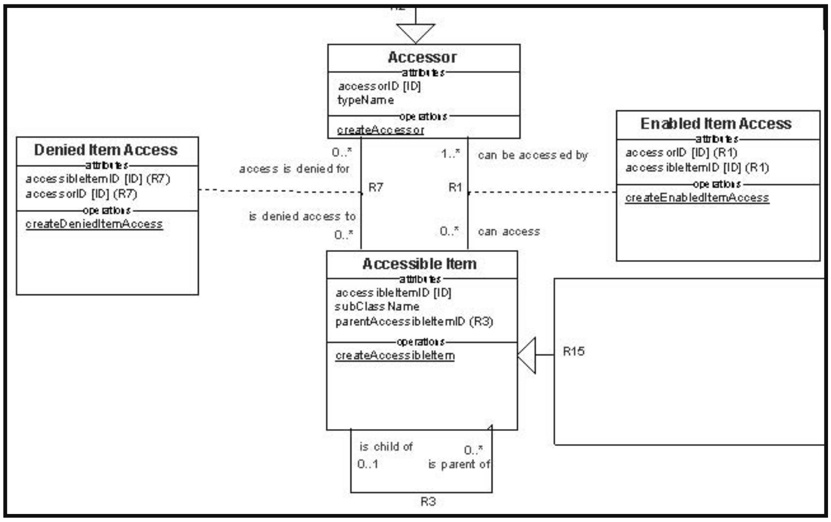


Fig. 7. Modelling access rules in data

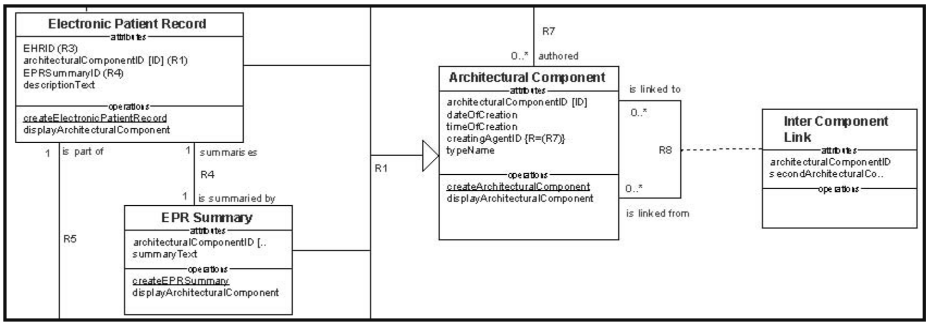


Fig. 8. Part of Patient Records domain class diagram

This counterpart relationship can be used at runtime to maintain an "anonymous" link between the two counterpart classes. The figure below illustrates how:

1. Whenever a new item of patient data is created in the "Patient Records" domain:
   - an object of "Architectural Component" is created;
   - a "counterpart" object of "Actual Accessible Item" is created;
   - a "counterpart" link (CPR2) is created between these two new objects
2. Whenever a legal access request is made on an "Actual Accessible Item" in the "Access Control" domain:
   - the counterpart relationship CPR2 is navigated to find the counterpart "Architectural Component";
   - the polymorphic operation "displayArchitecturalComponent" is invoked, which will in turn cause the appropriate rival method in the corresponding subclass to be executed for the corresponding subclass object.

The model fragment in Fig 10 below shows how this was achieved using a UML action specification language.

Note that the access rules are not hard-wired, but are captured as data in the form of objects, as illustrated in Fig 11 below:

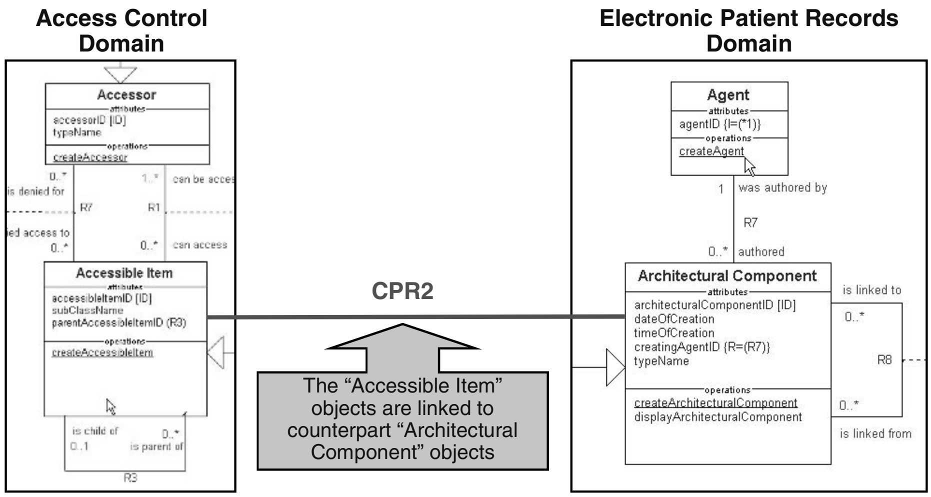
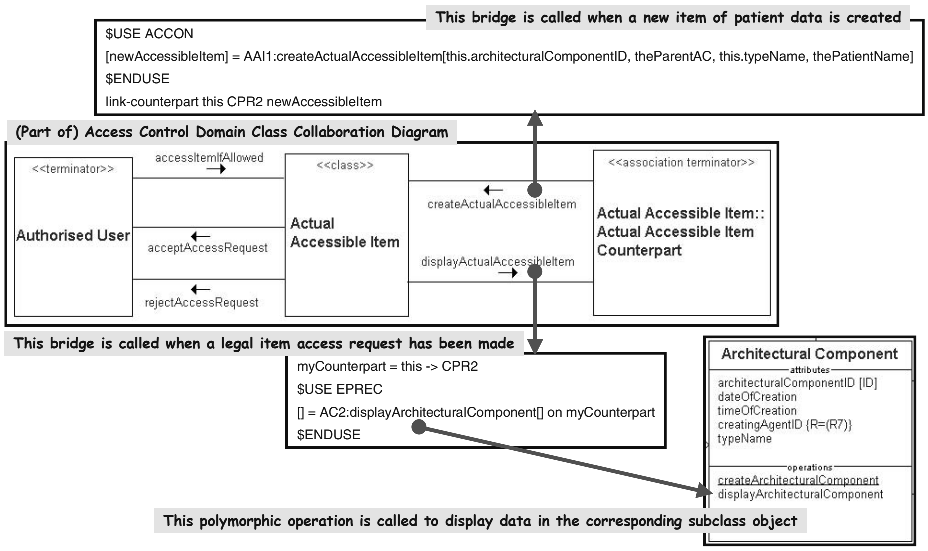


Fig. 9. A counterpart association, linking classes in separate domains


This bridge is called when a new item of patient data is created

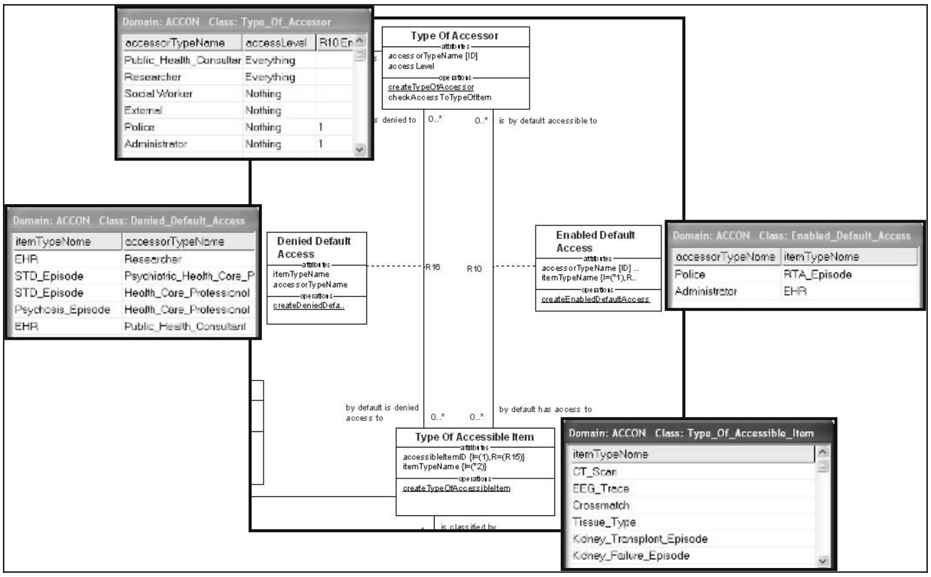
```
$USE ACCON
[newAccessibleItem] = AAI1:createActualAccessibleItem[this.architecturalComponentID, theParentAC, this.typeName, thePatientName]
$ENDUSE
link-counterpart this CPR2 newAccessibleItem
```

(Part of) Access Control Domain Class Collaboration Diagram
<<terminator>>
Authorised User
accessItemIfAllowed
acceptAccessRequest
rejectAccessRequest
<<class>>
Actual Accessible Item
createActualAccessibleItem
displayActualAccessibleItem
<<association terminator>>
Actual Accessible Item:: Actual Accessible Item Counterpart
This bridge is called when a legal item access request has been made

```
myCounterpart = this -> CPR2
$USE EPREC
[] = AC2:displayArchitecturalComponent[] on myCounterpart
$ENDUSE
```

Architectural Component
attributes
architecturalComponentID [ID]
dateOfCreation
timeOfCreation
creatingAgentID {R=(R7)}
typeName
operations
createArchitecturalComponent
displayArchitecturalComponent
This polymorphic operation is called to display data in the corresponding subclass object

Fig. 10. Bridge operations linking operations in separate domains



Domain: ACCON Class: Type_Of_Accessor

accessorTypeName	accessLevel	R10 En
Public_Health_Consultar	Everything	
Researcher	Everything	
Social Worker	Nothing	
External	Nothing	
Police	Nothing	1
Administrator	Nothing	1

Domain: ACCON Class: Denied_Default_Access

itemTypeName	accessorTypeName
EHR	Researcher
STD_Episode	Psychiatric_Health_Care_P
STD_Episode	Health_Care_Professional
Psychosis_Episode	Health_Care_Professional
EHR	Public_Health_Consultant

Domain: ACCON Class: Enabled_Default_Access

accessorTypeName	itemTypeName
Police	RTA_Episode
Administrator	EHR

Domain: ACCON Class: Type_Of_Accessible_Item

itemTypeName
CT_Scan
EEG_Trace
Crossmatch
Tissue_Type
Kidney_Transplant_Episode
Kidney_Failure_Episode

Fig. 11. Capturing access rules using data in "specification classes"

This makes the model very flexible, and easy to configure for different access rules that come about as a result of requirements and legislation changes. Page: 214 In effect the model is itself an example of a "domain specific language". The key notion here is that the form of the data that has to be entered maps very easily to the problem domain itself. In other words, the language used for configuration has been created to match the domain.

This technique meant that the Access Control domain could be easily integrated with any existing patient records component, if necessary by building a small wrapper that provided this interface.

3.1 Platform Independent UML Action Language

The "Access Control" component is intended to run on a wide variety of platforms across the UK. Platform independence was therefore critical, as it would not be feasible to build and maintain multiple versions of this component for each different platform.

UML allows construction of platform independent domain models that incorporate a UML action language. An action language makes a crucial contribution to achieving platform independence. Where UML modelling tools require a specific 3GL to be incorporated in the models in order to achieve some form of executability, platform independence is severely compromised.

The example in Fig 12 below shows a fragment of the "Access Control" domain, and the action language for the operation "Check for Denials" on the "Actual Accessible Item" class:

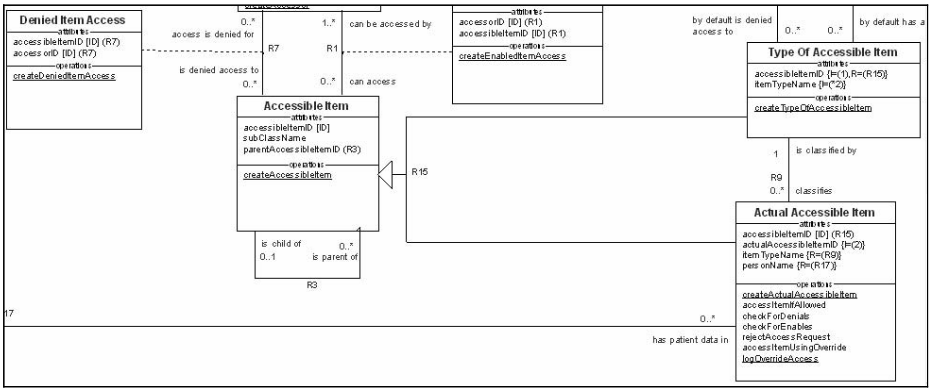


Fig. 12. Part of the "Access Control" domain

```
explicitDenial = TRUE

myAccessibleItem = this -> R15
myPIRAccessor = theAccessingPIR -> R2

# Is item explicitly denied to this PIR?
myDeniedAccess = myPIRAccessor and myAccessibleItem ->
   R7.Denied_Item_Access
if myDeniedAccess = UNDEFINED then
  #  PIR ok, what about Role?
  myRoleAccessor = theAccessingPIR -> R4.Role -> R2
  myDeniedAccess = myRoleAccessor and myAccessibleItem -
   >R7.Denied_Item_Access
  if myDeniedAccess = UNDEFINED then
    # Role ok, what about Person?
    myPersonAccessor = theAccessingPIR -> R4.Person -> R2
    myDeniedAccess = myPersonAccessor and myAccessibleItem -
    >R7.Denied_Item_Access
    if myDeniedAccess = UNDEFINED then

      # Everything OK so far - no explicit denials
        explicitDenial = FALSE
     endif
  endif
endif
```

This example segment of UML action language illustrates the benefits of using a language that operates at the level of abstraction of UML, rather than a 3GL such C++ or a 4GL such as SQL. Note that the modeller can simply state "myPersonAccessor = theAccessingPIR -> R4.Person -> R2" to navigate a chain of associations, without prejudice to how these associations are implemented.

3.2 Model Integration

3.2.1 Integrating with Diverse Existing Systems

It was necessary to deal with areas with localised differences in the processing across the external NHS organisations, such as the code to access patient data. This is addressed by a separate wrapper domain with a superclass representing the general (polymorphic) services to be provided, and subclasses encapsulating the site-specific code to perform each of those services. This isolates the “Access Control” domain from knowledge of the different processing schemes, and allows introduction of new schemes without creating a ripple effect.

3.2.2 Integrating with COTS Components

There were a number of candidate technologies to be integrated, such as PKI and Certification. These were dealt with by introducing "Interface Mapping" domains, illustrated in Fig 13, that map from a platform independent set of services used by the “Access Control” domain, to a platform specific set of services provided by the selected product(s). Examples of these are “PKI Interface” and “Registry Interface”. This reduces the ripple effect of introducing new products. The integration of the automatically generated “Access Control” components onto the various target environments would typically be achieved by building wrappers to map between the various component interfaces.

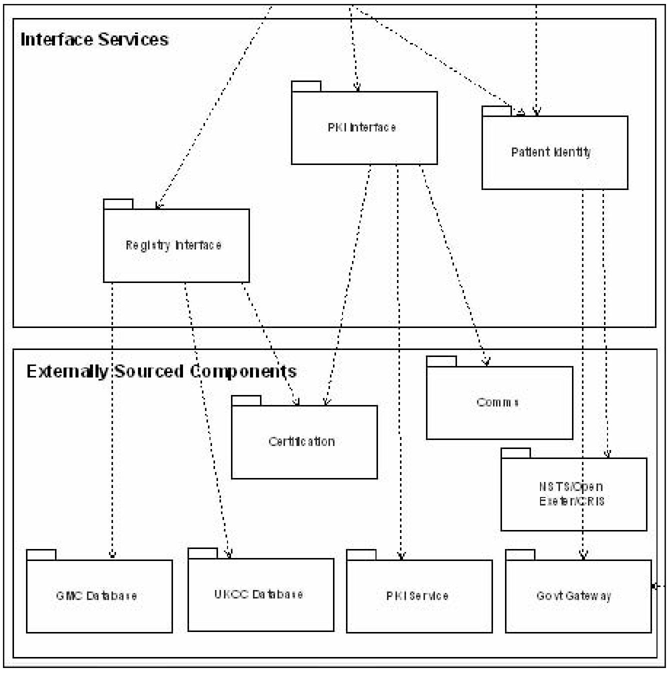
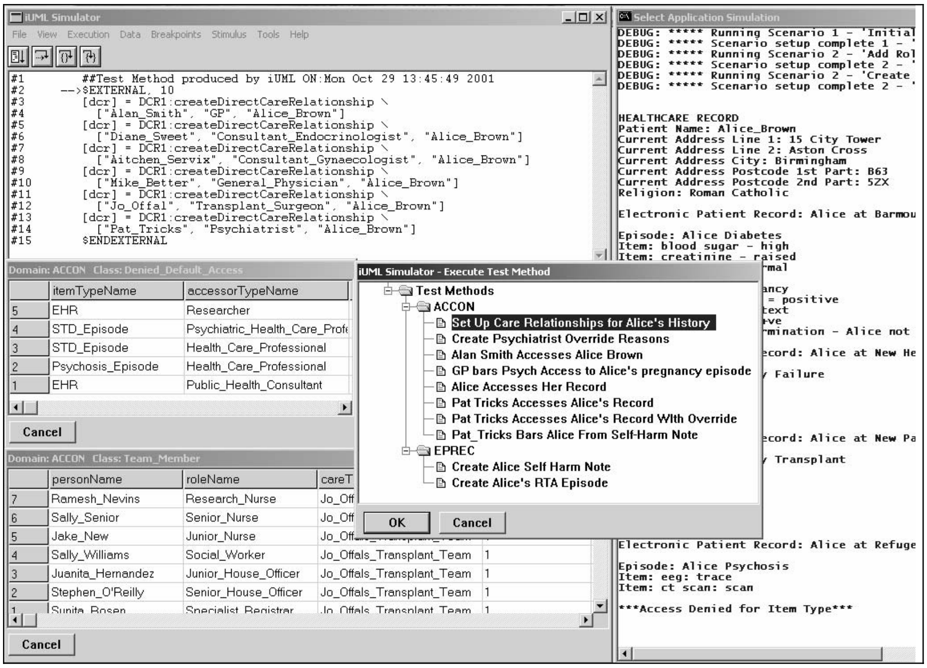


Fig. 13. Interface domains to decouple application from specific COTS products

4 Demonstration and Model Validation

xUML allows the interactive or batch testing of systems composed of single or multiple domain models. During testing, the user interacts at the UML model level. This allows rapid construction of stakeholder demonstrators, which can be iteratively refined in response to feedback from those stakeholders. Fig 14 shows an example of a PIM executing in a simulation environment.

It also allows development of a fully executable specification, free from the ambiguities that have caused problems in other similar systems, providing a strong basis for external procurement.


iUML Simulator
File View Execution Data Breakpoints Stimulus Tools Help
iUML Simulator - Execute Test Method
OK
Cancel
Select Application Simulation

```
      ##Test Method produced by iUML ON:Mon Oct 29 13:45:49 2001
  -->$EXTERNAL, 10
      [dcr] = DCR1:createDirectCareRelationship \
        ["Alan_Smith", "GP", "Alice_Brown"]
      [dcr] = DCR1:createDirectCareRelationship \
        ["Diane_Sweet", "Consultant_Endocrinologist", "Alice_Brown"]
      [dcr] = DCR1:createDirectCareRelationship \
        ["Aitchen_Servix", "Consultant_Gynaecologist", "Alice_Brown"]
      [dcr] = DCR1:createDirectCareRelationship \
        ["Mike_Better", "General_Physician", "Alice_Brown"]
      [dcr] = DCR1:createDirectCareRelationship \
        ["Jo_Offal", "Transplant_Surgeon", "Alice_Brown"]
      [dcr] = DCR1:createDirectCareRelationship \
        ["Pat_Tricks", "Psychiatrist", "Alice_Brown"]
      $ENDEXTERNAL
```

Domain: ACCON Class: Denied_Default_Access

	itemTypeName	accessorTypeName
5	EHR	Researcher
4	STD_Episode	Psychiatric_Health_Care_Profe
3	STD_Episode	Health_Care_Professional
2	Psychosis_Episode	Health_Care_Professional
1	EHR	Public_Health_Consultant

Domain: ACCON Class: Team_Member

	personName	roleName	careT	
7	Ramesh_Nevins	Research_Nurse	Jo_Off	
6	Sally_Senior	Senior_Nurse	Jo_Off	
5	Jake_New	Junior_Nurse	Jo_Off	
4	Sally_Williams	Social_Worker	Jo_Offals_Transplant_Team	1
3	Juanita_Hernandez	Junior_House_Officer	Jo_Offals_Transplant_Team	1
2	Stephen_O'Reilly	Senior_House_Officer	Jo_Offals_Transplant_Team	1
1	Sunita_Rosen	Specialist_Registrar	Jo_Offals_Transplant_Team	1

Test Methods
- ACCON
  - Set Up Care Relationships for Alice's History
  - Create Psychiatrist Override Reasons
  - Alan Smith Accesses Alice Brown
  - GP bars Psych Access to Alice's pregnancy episode
  - Alice Accesses Her Record
  - Pat Tricks Accesses Alice's Record
  - Pat Tricks Accesses Alice's Record WIth Override
  - Pat_Tricks Bars Alice From Self-Harm Note
- EPREC
  - Create Alice Self Harm Note
  - Create Alice's RTA Episode

```
DEBUG: ***** Running Scenario 1 - 'Initial
DEBUG: ***** Scenario setup complete 1 - '
DEBUG: ***** Running Scenario 2 - 'Add Rol
DEBUG: ***** Scenario setup complete 2 - '
DEBUG: ***** Running Scenario 2 - 'Create
DEBUG: ***** Scenario setup complete 2 - '


HEALTHCARE RECORD
Patient Name: Alice_Brown
Current Address Line 1: 15 City Tower
Current Address Line 2: Aston Cross
Current Address City: Birmingham
Current Address Postcode 1st Part: B63
Current Address Postcode 2nd Part: 5ZX
Religion: Roman Catholic

Electronic Patient Record: Alice at Barmou

Episode: Alice Diabetes
Item: blood sugar - high
Item: creatinine - raised
                     rmal

                     ancy
                      = positive
                     text
                     ve
                     rmination - Alice not

                     ecord: Alice at New He

                     y Failure




                     ecord: Alice at New Pa

                     y Transplant




Electronic Patient Record: Alice at Refuge

Episode: Alice Psychosis
Item: eeg: trace
Item: ct scan: scan

***Access Denied for Item Type***
```

Fig. 14. Executing the UML models for debugging and demonstration

Note that this form of simulation, with a relatively primitive textual user interface, was intended to provide stakeholder feedback on the capabilities of the access control component. The next stage is to integrate the newly engineered components into an HTML environment so that a more user-friendly interface can be demonstrated.

5 Conclusion

The use of MDA and executable UML provides a powerful solution to the problems of:

- developing new components to be integrated with a diverse set of existing components, where the strong emphasis on domain separation and interface specification simplifies the integration task;

- porting the new components to multiple platforms, where the use of **code generation from executable models** makes maintaining multiple platform-specific implementations feasible;
- establishing a stable and agreed set of requirements, where use of **UML simulators** to solicit feedback from stakeholders reduces the risk of delivering a system that does not meet user needs.

References

[1] Object Management Group - Model Driven Architecture - www.omg.org/mda
[2] S. Shlaer, S. J. Mellor. *Object Oriented System Analysis: Modelling the World in Data*. Yourdon Press Computing Series. (March 1988).
[3] S. Shlaer, S. J. Mellor. *Object Lifecycles: Modelling the World in States*. Yourdon Press Computing Series. (April 1991).
[4] S. J. Mellor, M. Balcer. *Executable UML. A Foundation for UML*. Addison-Wesley Pub Co; 1st edition. (May 2002)
[5] C. Raistrick et al. *Model Driven Architecture with Executable UML*. Cambridge University Press. (March 2004).

Managed Architecture of Existing Code as a Practical Transition Towards MDA

Nikolai Mansurov and Djenana Campara

Klocwork, 1 Chrysalis Way, Ottawa, Canada, K2G 6P9
mansurov@klocwork.com

Abstract. Managed Architecture is a practical, tool-assisted way of introducing modeling into projects at the evolution phases that work with existing code and do not have up-front models. By automatically extracting certain architecturally significant models (called Container Models) and then refactoring them to achieve sufficient level of abstraction, it is possible to increase the capability level of the organization by managing the architecture of the system instead of the code. We show that Managed Architecture can also facilitate further transition to Model-Driven Development.

1 Introduction

Current methodologies often overemphasize the importance of the initial phase of the complete *multi*-release, evolutionary production of software. All phases of the complete life-cycle, *except* the initial one, involve already *existing code*.

There is a vast amount of useful, deployed, operational software representing an enormous commercial value. Usually, the source code is the main artefact that evolves. Changes to existing code are required in order to:

- Add new functionality (65% of effort, according to [12])
- Adapt to new operating environments (18% of maintenance effort)
- Fix bugs (17%)

Changes to existing software can have different *magnitude*. Maintenance addresses changes of *small magnitude*. Activities like adding major new features to existing system, or modularization, go beyond regular maintenance. Changes of *increasing magnitude* are required in order to keep up with the changes in business requirements. Drastic changes to the existing software are often referred to as *modernization*. Modernization includes such activities as *porting* to a new platform, *migration* to a new technology, *migration* to COTS components, or *scaling* an existing system in order to increase performance or throughput, utilize more powerful hardware, including multiprocessor clusters. Total redevelopment can be viewed as the extreme case of modernization.

Changing existing code to add a new feature can be more difficult, than designing an equivalent feature at the initial phase, because all design decisions have to be *constrained* with existing decisions. Of course, the cost of redesigning an existing system from scratch is often prohibitive (or at least totally unattractive to decision-makers) due to the fact that the existing system has evolved to adjust to the business requirements, and it the only representation of them. There are some challenges, specific to maintenance and evolution of existing software, which are not present during development of new software:

- high-level design decisions need to be rediscovered
- impact of changes to useful features (the ripple-effect)
- higher volume of information, needed to be considered in order to understand the existing design or the impact of a change(understanding of legacy decisions in unfamiliar source code, also known as *program comprehension*)
- expertise "walks away" as key developers change jobs
- need for training new personnel
- multi-site collaborative environments, as for example maintenance is outsourced, or relocated between development centres within a large multinational corporation
- change resistance due to *architecture erosion* (usually it is progressively more difficult to change the system).

Maintenance and evolution of existing software (such as repair, adaptation and program comprehension) occupies significantly bigger portion of the entire life-span of a software system, than pure design activities (such as understanding requirements for a feature, designing the feature and implementing the source code). According to [13], in the early 70s for an average software project 60% of the total effort was spent on the initial design and implementation, and 40% - on the maintenance and evolution. However, in the late 90s, this distribution has changed quite dramatically: 10% for the initial phase, and 90% for maintenance and evolution.

Also, according to [11], starting from the 90s, more programmers are involved in working with existing software, than in "green-field" projects (see Table 1).

Table 1. Forecasts for numbers of programmers (in millions) and distribution of activities

Year	New projects	Enhancements	Repair	Total
1990	3 (43%)	3 (43%)	1 (14%)	7
2000	4 (40%)	4.5 (45%)	1.5 (15%)	10
2010	5 (35%)	7 (50%)	2 (14%)	14

The challenges of maintenance and evolution of existing software has been largely ignored by methodologists, tool vendors and standardization communities. There exists a significant gap in methodology and tool support between the "green-field" projects and "existing code" projects.

OMG's Model-Driven Architecture (MDA) [10] is a new development approach focused at *complete life-cycle of software production*, that can at least in the long run mitigate the above situation. MDA emphasizes:

- Modeling using UML, MOF, etc,
- Separation of platform-independent and platform-dependent concerns,

- Transformation of models (eventually – into the code),
- Paradigm shift from maintenance of the code to maintenance of the model and the transformation specifications.

MDA addresses the program comprehension challenge, since the upfront model created at the initial phase is used to generate the code. This shifts the emphasis away from trying to use the code as the source of knowledge about the system. MDA also addresses modernization challenges since new transformation rules can be used to generate code with desired properties, for example targeting a different platform.

However, existing code can become a *barrier* for adoption of the new development methodologies, including MDA. Only a fraction of real-world projects can benefit from the new methodology and the corresponding tools. These projects have to be in the pre-launching phase, or at the initial phase; while a larger number of projects already have existing assets and no models.

Current methodologies that emphasize formal modeling languages and advanced tool support often do not scale up to address the challenge of existing software. Upfront re-modeling of large existing systems is prohibitively expensive, therefore the potential benefits of modern methodologies and tools are not attainable for programmers, who have to deal with existing software. At the same time, tools for maintenance are usually built on an ad hoc basis, and have low adoption in industry.

In this article, we report on our experiences in introducing modeling into maintenance and evolution projects. Our approach emphasizes transition from managing software systems at the code level to managing architecture models, which in our opinion, provides a solid methodological foundation for the maintenance and evolution activities of the complete life-cycle.

Managed Architecture is a new development approach focused on evolution of existing software assets [4,6]. It involves the following activities:

- Extracting an Architecture Model from existing code
- Refactoring of the Architecture Model
- Using the Architecture Model for impact analysis and modernization planning
- Proactive enforcement of architecture integrity

This approach addresses the program comprehension challenge by using the architecture model to re-capture, preserve and accumulate the knowledge of the existing code, at the level appropriate for decision-making. It eliminates or slows down architecture erosion by visualization and impact analysis.

The main point of this article is to demonstrate that Managed Architecture can also kick-start transition into MDA, when the upfront model is not available.

The rest of the paper discusses the challenges of extracting models from existing code; introduces particular kind of models called Container Models; shows how to use Container Models to manage architecture of existing code, and how to leverage these models in the transition to Model-Based Development.

2 Extracting Models from Existing Code

There are several challenges in introducing modeling into "existing code projects" within the traditional approach to maintenance and evolution. In our opinion, the most promising is the non-invasive approach - extracting models from existing code and using them for the core maintenance and evolution activities.

There are several general reasons why models (and the corresponding tools) that are considered useful for the initial phase do not scale up to evolution phases. Modeling plays different roles at various phases of the complete life-cycle (see Table 2).

Table 2. Models at various phases of the complete life-cycle

Initial phase	Evolution phases
Model drives development of code	Model represents existing code
Model is more compact than the code	Model can become larger than the code
Model is refined	Model is abstracted away from code

This gap is created because source code is the artefact that evolves. Therefore, the biggest challenge is to extract models at sufficiently higher levels of abstraction than the source code. Other challenges include the following:

- The model should be *precise* (so that it can be used to formally reason about the system, and ultimately – about the source code)
- The model should have sufficient *scope* (so that it represents enough interesting aspects of the source code)
- The model should be *scalable* (so that the level of abstraction can be adjusted)
- The model should facilitate program comprehension
- The model should allow automatic update as the code changes
- The model should allow incremental manual transformations (refactoring)
- The model should be *actionable,* i.e. changes in the model should be easy to map into changes in the code

Models that are easy to extract from the source code are usually precise, model elements being close to code concepts; but such models have insufficient level of abstraction. Examples of such models are flowcharts, function call graphs, and class diagrams. A model is *scalable*, if its model elements can be *composed*, and still remain within the same modeling framework. For example, function call graph is scalable, since a composition of functions is a function; a class diagram is (at least potentially) scalable, since a collection of classes can be looked at as a bigger class. Scalability of the model is a very important factor since it is directly related to the level of abstraction of the model. The level of abstraction of the model should be sufficient for understanding the code by humans. In our experience, composition is key to achieving this objective. However, some models are *less actionable*, than others. For example, function call graphs abstract away too many details of the code, therefore it is difficult to map changes in this model back to code. On the other hand, class diagrams and other *structural models* are more actionable.

3 Container Models

In this section we describe the modeling approach that satisfies the above requirements. Klocwork "*Container Models*" [4] are structural models that focus on components and their interfaces. Container models support unlimited hierarchies of containers that consist of sub-containers. Container models support transformation of containers (for example, moving sub-containers between containers, splitting containers, creating larger containers, renaming containers, etc.) and preserve precision of the interfaces between containers during these transformations.

An initial container model is automatically extracted from the source code. Elements of the model have associations to the source code. Then the model is manually *transformed* to increase the level of abstraction and to remove any accidental dependencies between containers. Transformations preserve precise interfaces between containers as they become more and more architecturally significant [3,4]. Containers represent, for example, build packages, modules, subsystems and layers. At the bot tom of the hierarchy of containers are files and classes. Below that level there are primitive symbols for methods and class members, as well as individual procedures, variables, macros, types, etc.. Thus Container Models capture architecture views of the existing system (this usually involves some manual transformations of the automatically extracted model). The resulting model can be used for architecture analysis and management.

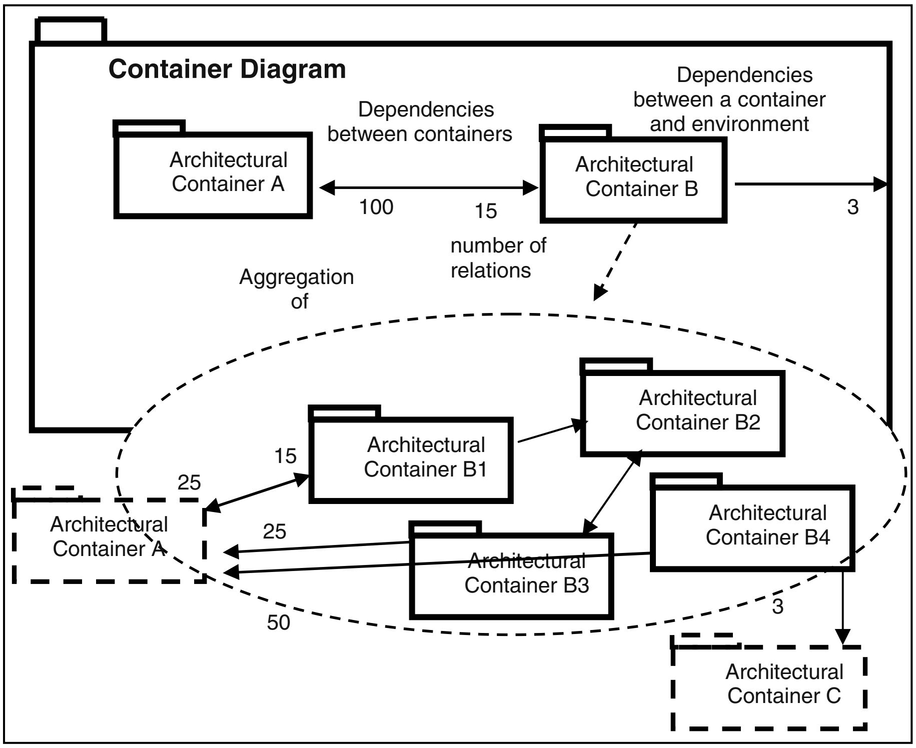

Container Diagram
Dependencies between containers
Dependencies between a container and environment
Architectural Container A
Architectural Container B
100
15
3
number of relations
Aggregation of
Architectural Container B1
Architectural Container B2
Architectural Container B3
Architectural Container B4
Architectural Container A
15
25
25
50
3
Architectural Container C

Fig. 1. Container models explained

Dependencies between containers represent "rolled-up" relationships between individual symbols that are placed into the corresponding containers. Whenever one set of symbols (for example, procedures, classes, files, directories, etc.) is grouped into one container and another set – into the second container, relations between individual symbols in different containers contribute to dependencies between the two containers. As the result, container diagram shows precise interfaces of each container.

Each relation is associated with a certain location in the source code. It is possible to navigate from any high-level container diagram directly to the source code level, for example to look for clues as to why a certain relation exists, and what is its responsibility. Klocwork Suite uses a flowchart graphical representation for viewing source code, independently of the programming language used [1,5]. It is also possible to investigate container interfaces and dependencies by navigating through dependency links and viewing the list of all individual items of the interface, or even the complete list of individual relations between symbols in each container.

Key operations on Klocwork Container Models are *composition*, *decomposition*, and *moving sub-components between containers*. Composition takes as input several sub-components at the same diagram, as well as the name of the new container, creates a new container at the current diagram and moves selected sub-components into the new container. Composition creates the new level of the hierarchy and dependencies between containers are recalculated. *Decomposition* is the opposite operation: it takes a container as input and removes it from the model while moving all its sub-components into the current diagram. Again, dependencies between containers are recalculated based on the relations between primitive symbols in containers.

Composition is an obvious way to raise the abstraction level of the model, however composition alone does not necessarily produce meaningful components because of various anomalies abundant in existing software. The key to meaningful architectural models is to *edit* the contents of containers by moving sub-containers and even individual symbols between containers.

Usually, we *move* individual symbols from one module to another. This implies a virtual editing of files although moving only occurs in the model. Moving operation allows eliminating dependencies between higher-level components that are induced by accidental placements of functionality at the file level. This happens far too often, when the physical architecture of the software is not managed properly. Moving sub-containers between containers is also possible, and desirable for refactoring of the Architectural Model.

Extraction of the Architecture Model from existing code is a *transformation* process that starts with the initial architecture model. There is no clear boundary between extraction of the architecture models, architecture analysis (what-if scenarios) and architecture optimization. Usually, extraction creates larger containers, and deals with the minor and accidental problems that prevent the understanding of the "real" architecture and therefore prevent the exposure of the "real" architecture problems due to the overwhelming complexity of the raw data. Transformations of container models can be defined as *architecture refactoring*, similar to design refactoring [16].

Extraction of a Container Model involves a large amount of *program comprehension* (including a mixture of domain expertise, generic software knowledge and common sense) in order to select families of symbols to compose and symbols to move to other containers. *Supplementary views* can facilitate such understanding. Supplementary views allow a temporary reduction of the complexity and therefore "unfreeze" the flow of the architecture extraction process. A supplementary view allows creating a temporary architecture view of all objects involved in a certain primary view. Usually this provides enough insight into the architectural abstractions involved. A partial architecture view can be efficiently extracted and then applied to the more complete context.

4 Managing Architecture Instead of Code

Managed Architecture is a new development approach focused at *evolution of existing software assets* [4,6]. It emphasizes:

- *extracting* an Architecture Model from code*,*
- *achieving* the sufficient level of abstraction by composition and Architecture Refactoring
- proactive *enforcement* of architecture integrity

The objective of this methodology is to create a formal architecture model of existing software that is high-level enough to be reasoned about and to be communicated to the development team. Therefore such model can be also used to manage the architecture of the existing software, to analyze problems with the current architecture, and to plan and coordinate the clean-up initiatives. This approach is particularly beneficial in a context where existing design and architectural documentation is absent, imprecise, or obsolete.

Architecture Models and Architecture Refactoring can significantly improve the software development capability of the organization. We demonstrate, that the architecture-centric tool support can help facilitate the transition from managing implementations to managing architectures [6].

What is a *managed architecture*? Managed Architecture is one of the advanced levels of the so-called Architecture Capability Maturity Model (ACMM). ACMM is a projection of the well-known SEI CMM model [18] to the domain of the software architecture. ACMM covers a single aspect of SEI-CMM – the architecture of the existing software [6].

Level 1: Initial Architecture
Any software has a certain *structure* - whether it is defined or not; understood or not. It consists of some *components* (maybe just a single monolith), components have some *dependencies* and are delivered as *configurations*.

Level 2: Repeatable Architecture
Often organizations use *repeatable patterns* for constructing and delivering software: some packaging rules, some use of libraries, some code reuse. Usually patterns start at the physical level (the *loadbuild*). Any organization that has large *software assets* involving *variation* (any embedded software falls into this category) becomes interested in *software architecture* issues, to determine success of the product line – the topic that has recently received much attention [17]. There are also some other drivers for organizations to become aware of their software architectures and to start *optimizing* them.

Level 3: Defined Architecture
At this level, components, their interfaces and configurations are formally *defined and maintained* together with the source code. Modeling tools like Rational Rose are used.

Usually, some component run-time framework is used. There are different scenarios as to when an organization moves to this level. Some may start from the “green field” and define their architecture before they start construction. Others do it later, when product line concerns become pressing enough [17].

However, the fact that an organization has defined the software architecture is the essential step to start managing the software from the architecture perspective.

Level 4: Managed Architecture
Software architecture is *manage*d when the organization understands the up-to-date, precise and quantifiable situation of the components, their dependencies and configurations. This requires several items:

- visualization of the architecture of existing software
- feedback between the "as designed" architecture and the "as built" architecture
- metrics of existing architecture
- use of the architecture model to understand the impact of changes
- organization-wide *enforcement of* the architecture integrity of the software.

Level 5: Optimizing Architecture
At this level, architecture integrity of the software is *enforced* and *improved*; the ongoing architecture improvement is part of the overall development process.

Managed Architecture level relies much more heavily on the use of software tools. The so-called modeling tools are used at Defined Architecture level in order to define, share and maintain the model in one of the formal modeling languages, e.g. UML. Tools that are used for Managing Architectures can be classified as Reverse Engineering or Software Quality Assurance tools. Such tools work with existing code to visualize, quantify and transform architecture of the existing code.

The architecture model can be used to enforce architecture integrity of the system. A typical *architecture authority role* is rather *reactive*: first some guidelines are formulated then they are given to developers to implements, then the results are evaluated. Usually, the construction cycle is the longest, so architecture faults have already happened and it is somewhat too late and costly to fix them. Therefore we often talk about *architecture erosion* - the uncontrolled degradation of the component dependencies over time when the architecture authority loosens his or her grip.

Tools for software architecture management shorten the cycle evaluation cycle. However, a proactive approach to managing software architecture should integrate integrity enforcement right into the construction process. For example, certain architecture rules should be checked on the fly right at the time when designer attempts to submit an updated module into the configuration management system.

5 Why Not Use UML for Managed Architectures?

Architecture Models for Managed Architecture are not UML for the following reasons:

- scalability, the need to maintain coherent and editable hierarchy of composed containers

- the need to handle large models at the initial extraction steps
- the need to refactor the architecture model while maintaining its precision
- specific "existing code" understanding concerns, like links to the code, navigation, etc.

While some of these requirements are tool-related, others are due to the differences in modeling vs. management concerns. Klocwork Container Models are based on UML package and object diagrams [2]. The underlying meta-model of Klocwork container models is significantly more complex than the meta-model for the corresponding UML package and object diagrams. The additional complexity is required to support unlimited hierarchies and mapping of primitive relations between symbols onto dependencies of containers at arbitrary levels of the hierarchy, as well as the requirement to keep associations to the source code [4].

On the other hand, each individual diagram in the hierarchy can be easily transformed into UML, as will be explained in the next section. This is consistent with the experience of other research groups, for example [14].

6 Jump-Starting Model-Driven Development with UML from Managed Architectures

The industry is starting to realize that the source code itself is not the right artifact for maintenance and evolution since it mixes platform-independent and platform specific concerns. The Model-Driven Architecture (MDA) approach, now standardized by OMG [10] promotes the shift from maintaining the source code to working with models and transformations that can derive the source code for the selected platform.

The key concept of MDA is the transition from maintaining the code to *modeling*. MDA also advocates separation of Platform-Independent Models (PIM) and Platform-Specific Models (PSM). The implementation code for selected platform is derived from the PIM through PSM with the use of automated code generation.

However, existing software introduces the so-called "*legacy barrier*" for transition to MDA, since it requires extraction of PIMs of existing software.

We believe that our container models that support unlimited composition, refactoring and links to the source code is a practical way to create architecturally significant models of existing software that can be gradually refined into PIMs for existing modules, and thus will allow integration of existing modules into the MDA–oriented software development. Container models correspond well to the models used during forward engineering, therefore they allow penetrating the "legacy barrier" for better adoption of advanced software engineering methodologies in industry.

Fig 2. illustrates the architecture-centric migration towards MDA. The left part of the figure illustrates the transformations, involved in MDA (PIM to PSM to code). These transformations assume existence of a detailed executable model. The challenge of integrating existing software systems into the "MDA world" is that such model usually does not exist and needs to be recovered in a cost-efficient way. In our opinion, the "straight-forward approach" of going from the source code directly to

PIM is not feasible because of the complexity of the existing software artifacts. Instead, we advocate the following architecture-centric approach.

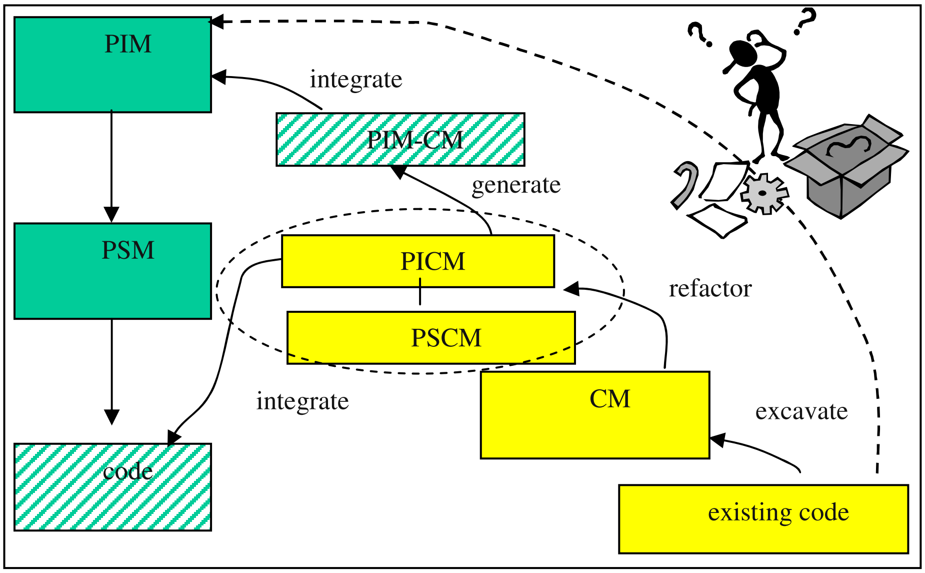


Fig. 2. Transition from Architecture Models (container models) to MDA models

Firstly, perform the transition to managed architecture by extracting the container model (CM) of existing code. As demonstrated earlier, this transition usually involves some architecture refactoring in order to regain intellectual control over the eroded architecture.

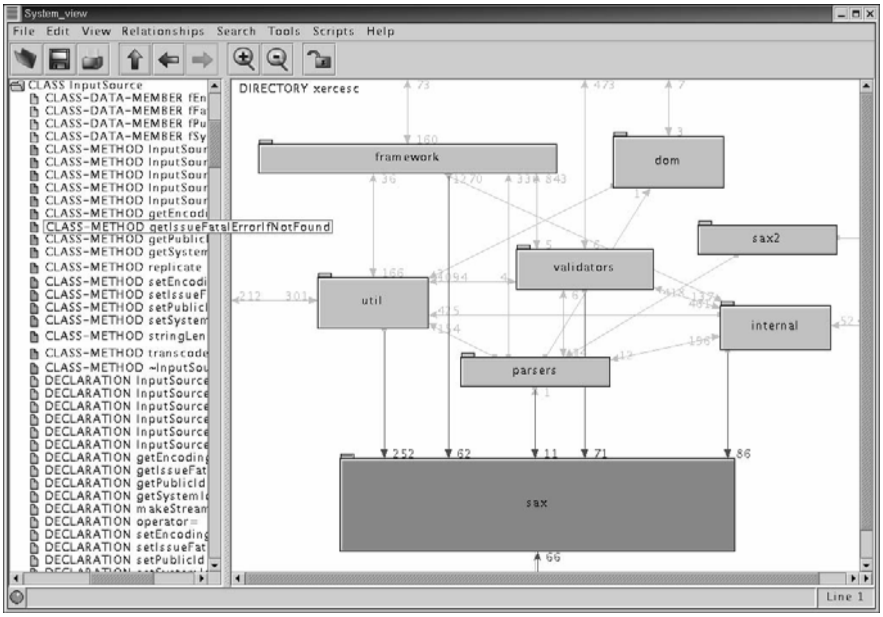
Secondly, refactor the model into two parts: Platform-Independent Container Model (PICM) and Platform-Specific Container Model (PSCM).

Thirdly, generate the UML component model, corresponding to the PICM and export it into MDA tools. New development involving the existing code base can now leverage this model. Thus an architecture-centric model of existing software can be integrated into MDA model. Integration of the existing and newly engineered components occurs at the code level, through managed interfaces.

Fig 3-4 illustrate a Container model (sax component of the xerces system) and the generated UML component diagram.

This architecture-centric model is not fully MDA-enabled, of course, since it only addressed components and their interfaces, but not their behavior. Existing components still need to be integrated with the new components, generated by the MDA tools.

The advantage of this approach is that the integrity of the model can be enforced using the tool support described earlier.



```
CLASS InputSource
  CLASS-DATA-MEMBER fEn
  CLASS-DATA-MEMBER fFa
  CLASS-DATA-MEMBER fPu
  CLASS-DATA-MEMBER fSy
  CLASS-METHOD InputSour
  CLASS-METHOD InputSour
  CLASS-METHOD InputSour
  CLASS-METHOD InputSour
  CLASS-METHOD InputSour
  CLASS-METHOD getEncodi
  CLASS-METHOD getIssueFatalErrorIfNotFound
  CLASS-METHOD getPublicI
  CLASS-METHOD getSystem
  CLASS-METHOD replicate
  CLASS-METHOD setEncodi
  CLASS-METHOD setIssueF
  CLASS-METHOD setPublicI
  CLASS-METHOD setSystem
  CLASS-METHOD stringLen
  CLASS-METHOD transcode
  CLASS-METHOD ~InputSou
  DECLARATION InputSource
  DECLARATION InputSource
  DECLARATION InputSource
  DECLARATION InputSource
  DECLARATION InputSource
  DECLARATION InputSource
  DECLARATION getEncoding
  DECLARATION getIssueFat
  DECLARATION getPublicId
  DECLARATION getSystemI
  DECLARATION makeStream
  DECLARATION operator=
  DECLARATION setEncoding
  DECLARATION setIssueFat
  DECLARATION setPublicId
```

Fig. 3. Example container model (sax component of xerces system)

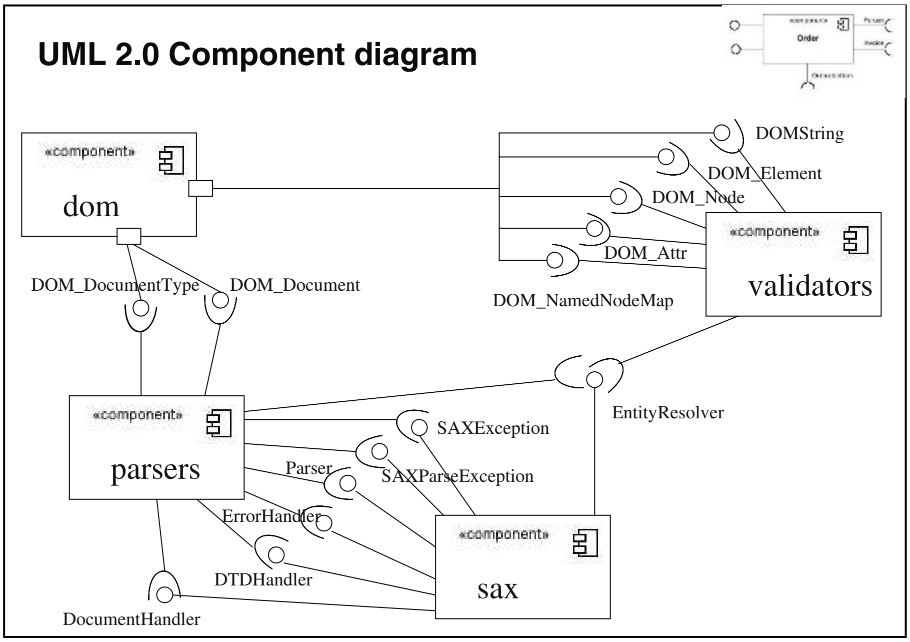


Fig. 4. Generated UML component diagram

7 Broader Context

We described our Managed Architecture approach and demonstrated, how it can enable transition to MDA. This approach is based on a number of pilot projects done by Klocwork and its customers over the last 3 years [15]. As the MDA tools mature, we expect to put more effort into supporting transition to MDA from Managed Architectures, while currently we focus on extracting architecture models and using them for maintenance and evolution of projects with existing code. The rest of this section discusses these activities in a larger context of standardization efforts within OMG.

7.1 Towards a Standard Meta-model for Representing Knowledge About Existing Software

Klocwork is leading the OMG Task Force on Architecture-Driven Modernization (ADM). A large industry of software tool vendors and service vendors exists to enable modernization of existing software [7,8]. There is a clear and urgent need for standardization to enable integration and interoperability among solutions from multiple vendors. Standardization will increase interoperability between different tools by creating an open framework. This can enable a new generation of solutions to benefit the whole industry [8].

Members of the ADM Task Force believe, that standardization of ADM models will help industry and individual businesses by *reducing the risk* of undertaking software improvement initiatives. The ability to share common information across projects that use a variety of tools and processes will lessen the time, risk and cost of software transformations. This will in turn improve the quality of reverse engineering and ADM tools, provide new capabilities, and extend the return-on-investment in software development tools. ADM standards will allow users to begin modernization projects knowing that there is interoperability between different tool vendors and that their solutions are extensible. Standardization will ensure that end users are investing not just in individual tools but rather into a coordinated strategy.

The first RFP of the ADM Task Force solicits proposals for a *Knowledge Discovery Meta-model* (KDM) for exchanging information related to transformation of existing software assets. Specifically, the RFP seeks a common repository structure to represent information about existing software and its operating environment [9]. KDM will provide the ability to document existing systems, discover reusable components in existing software, support porting to other languages and to MDA, or enable other potential transformations.

The meta-model will also enable information about existing software artifacts to be exchanged among different tools and be customized for specific end-user projects. This will enable vendors that specialize on certain languages, platforms or types of transformations to deliver customer solutions in conjunction with other vendors.

The KDM will represent the structure of the existing software and its related artifacts. It is important to represent all the principal artifacts of existing software, which in general terms can be described as the *entities* (the structural elements, the "things"), their *relations*, and *attributes* [9].

The KDM meta-model is not going to be restricted to a particular implementation language or platform. The biggest challenge is to represent behavioral programming artifacts in a language-independent (yet *actionable*) way. KDM will represent behavioral artifacts also as entities, relations and attributes. Entities at the language level include, for example, methods, source files, classes, screen definitions, data elements, records, tables or transactions. Relationships at the language level include, for example, "function uses a variable", "class inherits from another class", or "file includes a header". Attributes at the language level include, for example, name, access rights, version, source language, last scan date, or type of relationship.

The architecture aspect is key to ADM. At the architectural level there are several kinds of structures (corresponding to the well-known architecture views):

- Physical, or build structures (for example, files, directories, import relations between files, build dependencies, etc.)
- Run-time structures (for example, processes and threads and the corresponding interprocess communication channels, etc.)
- Logical structures (for example, subsystems, modules, layers, components and their dependencies, various architecture views, etc.)

Physical structures that are related to the build process of the existing software are often critical in ADM projects. Architectural entities include, for example, modules, subsystems, architecture layers, components or libraries. Architectural relationships include, for example, "file is contained in directory", "component provides a method", "component uses API of another component", etc. Architectural attributes include, for example, component name.

Klocwork is contributing their Container Models as the basis for KDM. For the purposes of interoperability, KDM will provide more specialized meta-classes to represent physical, run-time and logical structures. The meta-model will relate the elements to language-level structures to logical structures, run-time structures and physical structures through the mechanism of "rolled up" relations between containers, described in section 3.

8 Conclusions

We have discussed our approach to introducing modeling into "existing code projects". The objective of our approach is to use models to gain intellectual control over the architecture of existing software. The short-term benefits come from gradually improving architecture and maintenance robustness of existing software. The longer-term benefits may come from a unified, enterprise-wide use of model-based development "from the cradle to the grave".

In our experience, Container Models can facilitate successful introduction of modeling into "existing code projects". In summary, Container Models:

- Represent "containers", relations between "containers" and interfaces between "containers":
  - each "container" has dependencies on other "containers"
  - each "container" provides an API to other "containers"

- This model is *scalable*:
  - composition of "containers" is another "container"
  - The composition depends on everything that individual parts depended on
  - The composition provides the union of APIs, provided by individual parts
- Model can be *refactored*
  - Sub-containers can be moved from one container to another, model shows how dependencies and APIs change
- This model is *precise*
  - With respect to the contents of aggregations
  - With respect to APIs
- This model is meaningful and useful
  - Hierarchies of containers correspond to architecture views
  - Leaf "containers" can be procedures, variables, files, etc.
  - Leaf relations (APIs) are e.g. procedure-calls-procedure, etc.
- The Model is *actionable*: refactoring of containers and their contents can be mapped to edits of source files
- Model can be preserved and automatically updated as changes are made to software
  - Leaf containers and their relations are automatically extracted from source; the model stores only the hierarchy of containers; relations are recalculated on-the-fly

There are two distinct phases in our approach:

- Establishing Managed Architecture capability level
- Launch migration into MDA by generating UML models from container models

At the first phase, Managed Architecture techniques and reverse engineering tools are used to establish the following:

- Big picture of the system: major components and structures, relations between them
- Common context for documenting, developing and educating others about the system as it evolves, especially the legacy components
- Common repository for all architectural and design artifacts related to existing assets

At the second phase, MDA tools are used to perform the following:

- platform-independent modeling and validation
- platform-specific modeling
- Automatic code generation
- Common context for documenting, developing and educating others about the system as it evolves

At both phases, Managed Architecture techniques are used to enforce integrity of architecture cohesion of the system.

References

[1] N. Rajala, D. Campara, N. Mansurov, inSight Reverse Engineering CASE Tool, in Proc. of the ICSE'99, Los Angeles, USA, 1998.

[2] J. Rumbaugh, I. Jacobson, G. Booch, *The Unified Modelling Language Reference Manual*, Addison-Wesley, 1999

[3] N. Mansurov, A Systematic Approach to Recovering Architecture from Existing Software, SD Expo West, San Jose, 25th April, 2002:

[4] N. Mansurov, D. Campara, Extracting High-Level Architecture From Existing Code with Summary Models, in Proc. IASTED Conf. On Applied Informatics, Innsbruck, Austria, 2003

[5] N. Mansurov, D. Campara, Using Message Sequence Charts to accelerate maintenance of existing systems, in Proc. 10th SDL Forum, Copenhagen, 2001, LNCS, Springer Verlag, 2001:

[6] N. Mansurov , Using Metrics to enforce quality of Managed Architectures", in industrial presentations proc. of int. Conf. Metrics-2002, Ottawa, Canada, 2002:

[7] OMG, Legacy Transformation Working Group Forms, Draws Number of New OMG Members, OMG press release, July 2003

[8] OMG, Why do we need legacy transformation standards?, OMG whitepaper, 2003

[9] OMG Architecture-Driven Modernization: Knowledge Discovery Meta-model RFP, 2003.

[10] D. Frankel, et. al. *OMG Model-Driven Architecture*, Addison-Wesley, 2003

[11] A.van Deursen, P. Klint, C. Verhoef, Research issues in the Renovation of Legacy Systems, CWI reseaarch report P9902, April 1999

[12] I. Sommerville, *Software Engineering* (6th Edition), Addison-Wesley, 2000

[13] M. Lehman, Metrics and Laws of Software Evolution – The Nineties View, in Proc Metrics 97, Albuquerque, NM, 5-7 November 1997, pp. 20-32

[14] S. Ducasse, M. Rieger, S. Demeyer, Moose: An Extensible Language-Independent Environment for Reengineering Object-Oriented Systems, in Proc. 2nd Int. Symposium on Constructing Software Engineering Tools (CoSET 2000), June 2000

[15] Klocwork, http://www.klocwork.com

[16] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999

[17] P. Clements, L. Northrop, *Software Product Lines: Practices and Patterns*, Addison Wesley, 2002

[18] Carnegie Mellon University Software Engineering Institute, *The Capability Maturity Model: Guidelines for Improving the Software Process*, SEI Series in Software Engineering, Addison Wesley, 1995

EPTUD: An Eclipse Plugin for Testing UML Designs[⋆]

Trung Dinh-Trong[1], Nilesh Kawane[1], Sudipto Ghosh[1], Robert France[1], and Anneliese A. Andrews[2]

[1] Computer Science Dept., Colorado State University, Fort Collins, CO 80523
{trungdt, kawane, ghosh, france}@cs.colostate.edu
[2] School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164
aandrews@eecs.wsu.edu

1 Introduction

For model driven development approaches to succeed, there is a need for developing techniques for validating models. Studies show that many software faults occur in the design phase. Hence, it is essential to find and remove faults in design models. Currently, UML design models are typically evaluated using walkthroughs, inspections, and other informal types of design review techniques that are largely manual and consequently, tedious, error-prone and less effective.

We present an approach to testing UML design models consisting of class diagrams, sequence diagrams, and activity diagrams. Models under test are converted into an executable form that utilizes an underlying test infrastructure. The models are exercised with generated test inputs. We have implemented the approach as an Eclipse plugin (EPTUD — Eclipse Plugin for Testing UML Designs).

2 Test Approach

We assume that the diagrams are syntactically well-formed. This check can be done automatically by UML drawing tools. We also assume that the models under test describe sequential behavior only. We use the following types of actions in the activity diagrams: call operation actions, calculation actions, create and destroy object actions, create and destroy link actions, read and write link actions, and read and write variable actions.

Information from class and sequence diagrams is used to assess test adequacy. Information from class diagrams and activity diagrams is used to obtain the executable form of the design model.

[⋆] This research was supported in part by National Science Foundation Award #CCR-0203285 and an Eclipse Innovation Grant from IBM.

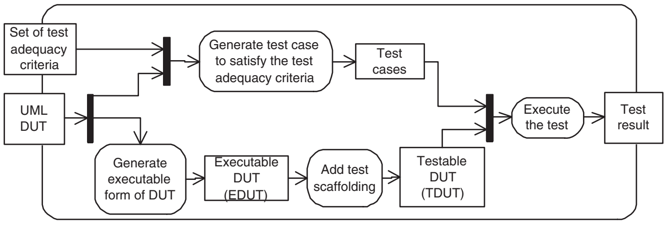

Set of test adequacy criteria
UML DUT
Generate test case to satisfy the test adequacy criteria
Test cases
Generate executable form of DUT
Executable DUT (EDUT)
Add test scaffolding
Testable DUT (TDUT)
Execute the test
Test result

Fig. 1. Overview of the Approach

The activity diagram in Figure 1 summarizes the overall testing process. Testing begins when a tester provides the UML design model under test, DUT, to the testing system and selects a set of test adequacy criteria [1].

A test case is a tuple consisting of three components: a prefix, P, a sequence of system events, E, and an oracle, O. Before a test is performed, the system is in an initial configuration containing a set of objects that can create any valid configuration of the DUT. The prefix, P, is a sequence of system events, which is applied to the system in the initial configuration to move it to the desired configuration in which testing can be started. Testing is performed by applying to the system a sequence of systems events, $E =< e_i : i = 1 \ldots n >$, where e_i is a system event. We restrict system events to be operation calls. The oracle, O, is used to define the expected behavior of the system. In our approach, an oracle is a sequence of tuples (o_i, e_i), where o_i is a condition (expressed in OCL) that the runtime configuration of the DUT must satisfy after the system event, e_i, is executed.

The DUT is converted into an executable form, $EDUT$, using design information in structural (class diagrams) and dynamic (activity diagrams) views of the design to simulate the behavior of the model. Test scaffolding is added to the executable form to automate test execution and enable runtime failure detection. The combination of the executable form of the design and the test scaffolding is called the testable form, $TDUT$.

Testing is performed by executing the $TDUT$ with the generated test inputs. During test execution, the effects of system behaviors modeled by activity diagrams are observed in terms of changes in the configurations. The configuration of the $TDUT$ is updated continuously during the test. Also, after the execution of each system event, e_i, the corresponding oracle condition, o_i, is checked. If a configuration produced during the test violates any constraint described by the class diagrams, or if any condition, o_i, evaluates to false, a failures is reported.

3 Eclipse Plugin

We have developed a prototype implementation of our approach in the form of an Eclipse plugin.

Specification of the Model Under Test: The Omondo EclipseUML plugin is used to draw class and sequence diagrams. Operation behaviors are described as actions using the Java-like Action Language (JAL) [2] developed by our research group. Developers use the ecore system editor to specify operations and OCL constraints.

Generation of the Executable and Testable Forms: UML classes, attributes, and operations are transformed into Java classes, state variables and method declarations. For each class, *C*, in the *DUT*, a collection class, *SetOfC*, is generated. An instance of *SetOfC* maintains a collection of instances of *C*. The *SetOfC* class is needed to take care of association-end multiplicities that are greater than 1. The *SetOfC* class has methods to add (or remove) an instance of *C* to (or from) the collection. Association ends are transformed into Java attributes with collection class types. For more details on transforming UML class diagrams into Java, please refer to Dinh-Trong [3].

A class named *TFactory* is generated from the class diagrams. This class has public methods to create and destroy instances of every class and association in the class diagrams.

Activity diagrams are transformed into Java method bodies using the following rules:

1. *Call* actions become Java method invocations.
2. Return actions become return statements.
3. *Create object* actions become Java object creation statements.
4. Java condition (`if` ... `then` ... `else` ...) and loop structures (`while` ...) are derived from activity condition and iteration structures respectively.
5. Object (or link) create and destroy actions are transformed into appropriate invocations of the methods in *TFactory*.

Scaffolding is added to the *EDUT* to obtain the *TDUT*. Scaffolding includes test drivers and code to detect test failures. Test drivers consist of Java code to (1) create the initial configuration, (2) apply test inputs to the system, and (3) execute tests. Failure detection involves execution of code that checks for certain failure conditions. The following conditions are checked:

1. Uninitialized variables in conditions (such as transition guards in activity diagrams).
2. Uninitialized parameters passed in operation calls.
3. Non-existent target object of an operation call.
4. Pre-conditions before method execution evaluate to false.
5. Post-conditions after method execution evaluate to false.
6. Object configuration produced by the execution of a system event violates constraints imposed by a class diagram.

The first three checks are performed by code inserted in the *EDUT*. For the last three checks, we use the facilities provided by the USE tool [4]. USE is an open source tool that validates whether a configuration conforms to the constraints described in a class diagram. USE accepts UML class diagrams in its own format. Therefore, EPTUD transforms the *DUT* into USE format.

Test Execution and Failure Reporting: Testing is performed by executing the *TDUT* using the generated test inputs. During test execution, the effects of system behaviors modeled by activity diagrams are observed in terms of changes in the configurations.

EPTUD provides USE with pre- and post-conditions specified in the OCL and requests USE to validate them for every operation before and after its execution respectively. Also, after the execution of every system event in the test input, EPTUD signals USE to check the object configuration against the class diagram constraints.

Because the tools perform a different set of failure checks, both maintain their own copies of the configuration during test execution. When testing begins, EPTUD signals USE to create its representation of the initial configuration. Whenever the configuration changes, USE is informed about the modification. The changes in the configuration include adding or removing an object or a link, and modifying an attribute value.

The configuration of the *TDUT* is updated continuously during the test. If a configuration produced during the test violates any constraint described by the class diagrams, a failure is reported.

4 Conclusions and Future Work

We outlined a systematic approach to testing UML design models and described an Eclipse plugin that supports the approach. We are currently adding capabilities to visualize test execution through animated sequence diagrams and observe test coverage in the different views of the *DUT*. Future work also includes developing techniques for test input generation.

References

1. Andrews, A., France, R., Ghosh, S., Craig, G.: Test Adequacy Criteria for UML Design Models. Journal of Software Testing, Verification and Reliability **13** (2003) 95–127
2. Dinh-Trong, T.T., Kawane, N., Ghosh, S., France, R.B., Andrews, A.A.: A Tool-Supported Approach to Testing UML Design Models. Technical Report CS 04-108, Computer Science Department, Colorado State University, Fort Collins, CO 80523 (2004)
3. Dinh-Trong, T.T.: Rules For Generating Code From UML Collaboration Diagrams and Activity Diagrams. Master's thesis, Colorado State University, Fort Collins, Colorado (2003)
4. Gogolla, M., Bohling, J., Richters, M.: Validation of UML and OCL models by automatic snapshot generation. In: Proceedings of the 6th Int. Conf. Unified Modeling Language (UML'2003), Springer, Berlin, LNCS 2863 (2003) 265–279

Towards a Platform for Debugging Executed UML-Models in Embedded Systems

Philipp Graf, Clemens Reichmann, and Klaus D. Müller-Glaser

Institut für Technik der Informationsverarbeitung, Universität Karlsruhe (TH), Engesserstr. 5, 76131 Karlsruhe, Germany
{graf, reichmann, mueller-glaser}@itiv.uni-karlsruhe.de

Abstract. Automated transformations from model to executable code require new appropriate model-based ways for debugging and monitoring such models directly on an embedded target platform. We propose an architecture that allows the definition of various debugging-scenarios and -views independent of the actual execution-platform. This architecture is utilized to explore new approaches for assisting the developer in finding faults in the system he is developing.

1 Debugging in the Context of Model Based Development

Model-based development is a natural progression in the search for higher level design-constructs to master the challenges of ever increasing complexity of the systems under design. A further aspect of abstracting the design process to higher level models is omitting information that is not known at this point of development, namely information specific to the deployment platform. Platform specific information is to be added during a final transformation phase to a model or directly to code. The OMG is following this idea with their Model Driven Architecture [1], but in principle this has always been accomplished by code-generators for various CASE-tools before.

It is a known fact that methods and tools for finding defects in software have always followed up new design- and programming-paradigms with a certain delay [2]. Nevertheless debugging takes up a huge amount of time, regardless of the used design- or programming-language. Development methodologies or higher-level constructs can reduce the number of faults, but never remove them completely.

Debugging largely remains a task that requires assistance by the developer. If the developer specified the system for instance using the UML and afterwards transformed the model to executable code automatically, debugging using a common source-code debugger will not give the appropriate view on the system. For example, a simple state-chart is typically transformed to several hundreds lines of code. This is even more critical, as the developer is not familiar with the automatically generated code.

Simulation also is not always an alternative, as especially for embedded systems timing-constraints have to be met. Also such systems usually interact heavily with their environment and peripheral hardware, such that writing simulators is inefficient and error-prone in itself.

This work has to be seen in the context of a model-based approach implemented in a tool called *GeneralStore*, where systems can be managed in a UML-centric way, but with subsystems defined in other notations. The models of these so-called domains can be transformed to executable code using commercial generators and our own approach for UML models. The interoperability between those system parts can be modeled using the UML and are automatically transformed to actual wrapper-classes during code generation. An in-depth explanation is given in [3] and [4].

2 Approach and Platform

We propose an approach of debugging running software using methods and views that are appropriate to the modeling notation used. The used architecture is shown in Fig. 1.

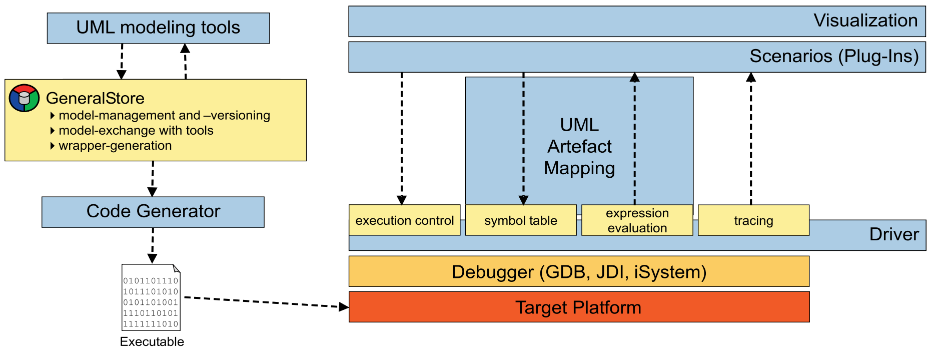


Fig. 1. Architecture of the platform for model-debugging

As described in [4] we allow for modeling a system building it from sub-models using multiple notations and meta-models. However at the current stage of development we only consider the debugging of UML models. Thus, the process starts with the system modeled using a UML CASE-tool. Our model-repository *GeneralStore* is capable of generating source code from UML Class Diagrams and UML 1.5 Actions (see [5]) and building an executable. As we focus on embedded systems, we generate C/C++, but generation of Java is also implemented.

The executable is run on the target platform. This can be any platform that exposes an interface for debugging, such as the GNU Debugger (GDB) [6], a JAVA virtual machine through JDI [7] or any interface for a hardware-supported debugger as they are common for embedded systems. These debugging interfaces and their APIs are mostly proprietary and very manifold. To abstract from the actual target we define a driver layer that wraps the various debugger-implementations on source-level. Each of these drivers implements an adapter for a set of interfaces it exposes to the layers on top. This is necessary as not all debuggers exhibit the same set of features.

Viewing the shown architecture from top there are various so-called scenarios that implement debug-features necessary to monitor and debug the system. These scenarios transform system-state or system-trace to various diagrams that are exposed to the user via the visualization-layer. Every scenario provides the visualization layer with a diagram model based on the UML Diagram Interchange, so that standardized serialization of visualization data is possible.

The UML Artefact Mapping layer is responsible for mapping artefacts from the metamodel used for modeling to artefacts (i.e. variables, lines of code,...) on source-code level. If the transformation to code followed the MDA pattern, we have to retransform queries using the same MDA mapping. Classical code-generators need a different mapping. Thus, we need to provide a mapping for every modeling domain, every type of transformation (e.g. MDA mapping, code-generator) and every transformed artefact. Note that this mapping is neither necessarily surjective nor injective or even a function.

The mapping itself is accomplished in two stages. The lower layer allows to transform queries on model level to queries on source level, thus abstracting from the used transformation. Atop this layer we provide an implementation of a query for every considered modeling artefact (e.g. attribute, association, state,...).

3 Prototype and Future Work

To verify the feasibility of this approach we are in the process of implementing a JAVA prototype of this architecture. It allows to monitor instances of classes and their relations in an executed model using UML Object Diagrams.

Fig. 2 shows the design flow for the implemented scenario. Prerequisite is a UML class-model that is imported into *GeneralStore*. The model is used to generate the runnable executable as described in [4]. This prototype is suitable for any platform that supports the GCC-toolchain. The debuggee is executed through GDB and stopped at a break-point defined on the class-model.

The object model is built starting from any object, usually through a static reference. We iterate over all attributes, regenerate the target-code for the required expressions and evaluate these querying GDB. The obtained values are remapped to the respective model data-type and added as a slot. Similarly we iterate over all associations and build missing links and recurse into generating linked object if required.

To allow presentation with our DI-viewer, we build diagram information from the model using fully automated layout and routing. The diagram is presented in Eclipse based on the Graph Editor Framework (GEF). Currently we are evaluating an integration with the TopModL-initiative [8], as their approach partly addresses infrastructure our debugger could benefit from.

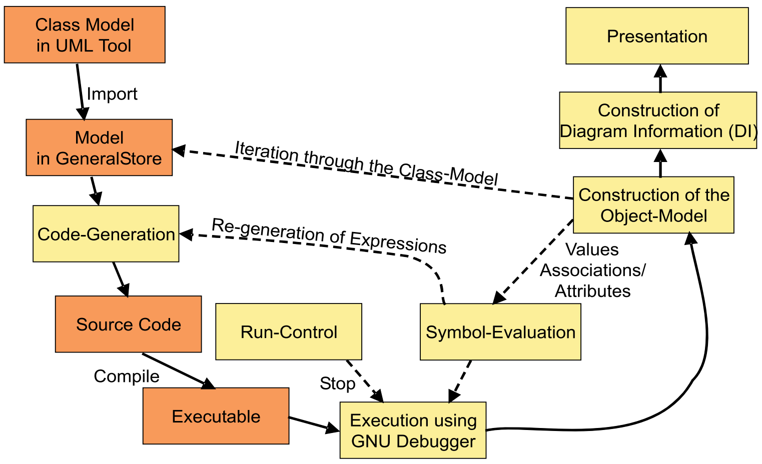

Class Model in UML Tool
Import
Model in GeneralStore
Iteration through the Class-Model
Presentation
Construction of Diagram Information (DI)
Construction of the Object-Model
Code-Generation
Re-generation of Expressions
Values Associations/ Attributes
Source Code
Run-Control
Symbol-Evaluation
Compile
Stop
Executable
Execution using GNU Debugger

Fig. 2. Design flow for visualizing program state with object diagrams

Future work will also focus on a more generic implementation. The visualization layer needs additional work, with more advanced and incremental layout and routing. Having built the platform, it will be possible to work on more scenarios which then have to be integrated into an actual tool sufficient for debugging models. Also cross-metamodel debugging for heterogeneously modeled systems has to be considered an important topic.

References

1. Object Management Group (OMG): MDA Guide Version 1.0.1 (2003)
2. Lieberman, H.: Introduction to The Debugging Scandal and What to Do About It. Communications of the ACM **40** (1997) 26–29
3. Kühl, M., Reichmann, C., Müller-Glaser, K.D.: From object-oriented modeling to code generation for rapid prototyping of embedded electronic systems. In: Proceedings of the IEEE International Workshop on Rapid System Prototyping (RSP 2002), Darmstadt, Germany (2002) 108–114
4. Reichmann, C., Kühl, M., Graf, P., Müller-Glaser, K.D.: Generalstore - a case-tool integration platform enabling model level coupling of heterogeneous designs for embedded electronic systems. In: Proceedings of the 11th IEEE International Conference on the Engineering of Computer-Based Systems, Brno, Czech Republic, Springer (2004)
5. Object Management Group (OMG): Unified Modeling Language (UML) Specification, Version 1.5 (2003)
6. GNU: The GNU Debugger. (http://www.gnu.org/software/gdb/)
7. Sun: Java Platform Debugger Architecture. (http://java.sun.com/products/jpda/)
8. The TopModL Initiative: TopModL Framework. (http://www.topmodl.org)

The TopModL Initiative

Pierre-Alain Muller[1], Cédric Dumoulin[2], Frédéric Fondement[3], and Michel Hassenforder[4]

[1] INRIA, Rennes, France
pa.muller@uha.fr
[2] LIFL, Université de Lille, France
cedric.dumoulin@lifl.fr
[3] EPFL/IC/LGL, Lausanne, Switzerland
frederic.fondement@epfl.ch
[4] MIPS, Université de Haute-Alsace, France
m.hassenforder@uha.fr

Abstract. We believe that there is a very strong need for an environment to support research and experiments on model-driven engineering. Therefore we have launched the TopModL project, an open-source initiative, with the goal of building a development community to provide: an executable environment for quick and easy experimentation, a set of source files and a compilation tool chain, and a web portal to share artifacts developed by the community. At the time of writing we have almost completed the bootstrap phase (known as Blackhole), which means that we can model TopModL and generate TopModL with TopModL.

1 Introduction – About Model-Driven Engineering

At the end of the year 2000, the OMG proposed a radical move from object composition to model transformation [1], and started to promote MDA[2] (Model-Driven Architecture) a model-driven engineering framework to manipulate both PIMs (Platform Independent Models) and PSMs (Platform Specific Models). The OMG also defined a four level meta-modeling architecture, and UML was elected to play a key role in this architecture, being both a general purpose modeling language, and (for its core part) a language to define metamodels. While preeminent in the current days, MDA is only a specific case, and we suggest considering model-driven engineering as a wider research field, which includes the study of the following issues:

- What are the essential entities and operations for model-driven engineering, and how to classify these entities and operations?
- How to separate and merge the business and platform aspects, and then how to build transformation systems and how to translate models into executable code?
- How to maintain a model-driven application and how to migrate a legacy application to a model-driven application, and how to integrate conventional applications with model-driven applications?
- Which abstractions and notations should be used to support the previous points and what kind of supporting environment should be defined?

Obviously the scope of model-driven engineering is wide and a lot of work is still ahead of us. We believe that a common research platform which would provide the fundamental services required by model-driven engineering would significantly contribute to the advance of research in this field. In this short paper we will focus on the presentation of the bootstrap phase of TopModL. We first briefly state the basic principles of model-driven engineering which shape the requirements of TopModL, and then move to the description of the bootstrap process, including an overview of the model transformations that we have developed to be able to reuse existing MOF-based metadata repositories. Then we present some related work and draw final conclusions.

2 Basic Principles of Model-Driven Engineering

The fundamental model-driven engineering principles identified so far by the TopModL initiative are:

- The fact that everything is a model. For TopModL, models are first-class entities; everything is expressed explicitly in terms of models, including business models, platform models, executable models, debugging models, trace models, transformation models, process models…
- The notions of languages, models and roles. A model is expressed in a language; this language is a model which plays the role of meta-model for the models expressed in that language.
- The independence versus the model repository. We want to have a uniform access to several repositories including EMF [3], MDR [4], or XDE.
- The fact that TopModL itself is model-driven. We want everything in TopModL to be explicit and customizable, including the meta-modeling framework. For instance TopModL does not require the M3 to be MOF, it does not even require a 4 layer meta-modeling architecture.

The basic services offered by TopModL include:

- Model (meta-model) persistence, serialization in XMI (XML Meta-data Interface) and manipulation via JMI (Java Meta-Data) interfaces.
- Visual edition of models (meta-models) with OCL evaluation.
- Model-Driven parameterization of TopModL and textual and visual editor generation.
- Model transformation and code generation (Java, SQL).

3 Blackhole – Technical Architecture of the Bootstrap

One of the major concerns of TopModL is to make explicit the meta-modeling framework, and not to constrain the usage of a meta-modeling language at the M3 modeling level. There are mainly two reasons for that. The first reason is that TopModL, as a research-oriented tool, should be able to evaluate new possible meta-modeling languages and new meta-modeling constructs. The second reason is that there are already many different meta-modeling languages available today, and that there will be even more in the future (MOF 1.3, MOF 1.4, MOF 2.0, EMF ECore, …), and so TopModL should be able to deal with all of them. As an example, the first M3 language that we want Blackhole to support is the UML 2.0 Infrastructure.

The first step is to model the meta-modeling language (the UML 2.0 Infrastructure in our case) using UML class diagrams in a UML CASE tool such as the Community Edition of Poseidon [5]. Then, this metamodel is exported in an XMI file as a UML 1.4 model, which can be taken as input by the MDR *UML2MOF* model transformation, which promotes UML models to MOF models. However a metamodel like the UML 2.0 Infrastructure is making an extensive use of constructs that do not have their counterpart in MOF 1.4 (like package merge or refinement/subsetting of association-ends). Therefore we have extended MOF 1.4 and defined MOF 1.4++ which provides support for these constructs. Then a model transformation translates the MOF 1.4++ model back to a MOF 1.4 model, using generalizations and redefinitions to implement the merges. Alike, association-ends are marked as derived, and a specific implemen tation is generated. In the end, the Infrastructure metamodel now conforms to MOF 1.4 and can be sent to a JMI generator (like MDR) to generate the repository and associated class interfaces and implementations.

We have also developed a visual editor which connects to this repository, and we can this way edit metamodels (which conforms to the UML Infrastructure). This editor is currently hand-coded, but it is our intent to generate it from models, in a similar way of what is done in Netsilon [6].

At this point, TopModL is able to edit and maintain Infrastructure models, thus it is possible to model the Infrastructure in TopModL. The result is the Infrastructure metamodel modeled as an Infrastructure model. Another model transformation was developed to transform Infrastructure models to MOF 1.4++ models, so that the Infrastructure metamodel modeled in TopModL can be automatically translated to a MOF 1.4++ model. By reusing this MOF 1.4++ model of the Infrastructure, it is possible to perform again the process, this time using TopModL instead of Poseidon: TopModL is at this phase bootstrapped, i.e. TopModL is modeled and designed using TopModL. Obviously, the bootstrapping process may be repeated for any modeling language supposed to play the role of a meta-modeling language.

4 Related Works

There are many related works, which share a common vision with the TopModL initiative. We summarize some of these approaches below:

- Meta CASE tools including Metaedit+ [7], Dome [8] or GME [9], provide customizable CASE tools, however they are fairly closed in the sense that they are either not open-source, or not themselves model-driven.
- Dedicated model-driven tools, which generate specific applications, like Netsilon [6] for Web information systems, Accord/UML [10] for embedded distributed real-time systems.
- OCL based tools, including KMF [11] which generates modeling tools from the definition of modeling languages expressed as meta-models, or Octopus [12] an Eclipse plug-in OCLE [13] or the Dresden OCL toolkit [14], which are able to check the syntax of OCL expressions, as well as the types and correct use of model elements like association roles and attributes.

- Meta-modeling frameworks like Eclipse EMF [3], Netbeans MDR [4] or Coral [15] which offer model persistence, model serialization and programmatic access to models via an API, or interoperability technologies like ModelBus [16]. These frameworks provide part of the functionalities required by TopModL.
- Open-source modeling tools, including ArgoUML [17] Pampero [18] or Fujuba [19] which offer significant features at the M1 level, but lack customization at the M2 level.

One of the goals of TopModL is to understand how to reuse or leverage these related works, and to find how to integrate them as much as possible in a research platform for model-driven engineering.

5 Conclusion

The TopModL open-source initiative has been launched with the goal of providing tool support to the model-driven engineering research community. TopModL groups a development community, a web portal to share the artifacts developed by the community, a set of source files and an executable program for meta-modeling.

References

[1] J. Bezivin, *"From Object Composition to Model Transformation with the MDA"*, in proceedings of TOOLS'2001. IEEE Press Tools#39, pp. 350-354 . (August 2001).

[2] Object Management Group, Inc., *"MDA Guide 1.0.1"*, omg/2003-06-01, June 2003.

[3] Eclipse EMF, web site http://www.eclipse.org/emf/

[4] Netbeans MDR, web site http://mdr.netbeans.org/

[5] Poseidon web site http://www.gentleware.de

[6] P.-A. Muller, P. Studer, and J. Bezivin, "*Platform Independent Web Application Modeling"*, in P. Stevens et al. (Eds): UML 2003, LNCS 2863, pp. 220-233, 2003.

[7] R. Pohjonen, "*Boosting Embedded Systems Development with Domain-Specific Modeling*", in RTC Magazine, April, pp. 57-61, 2003

[8] Honeywell, 1992, "*DOME Guide"*, available from "www.htc.honeywell.com/dome/"

[9] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, "*The Generic Modeling Environment"*, WISP'2001, Budapest, Hungary, may 24-25, 2001

[10] S. Gérard, N. S. Voros, C. Koulamas, and F. Terrier, "*Efficient System Modeling of Complex Real-Time Industrial Networks Using the ACCORD UML Methodology*", DIPES'2000,Paderborn, Germany, 2000.

[11] Kent Metamodeling Framework, web site http://www.cs.kent.ac.uk/projects/kmf/ index. html

[12] Octopus web site http://www.klasse.nl/ocl/octopus-intro.html

[13] OCLE web site http://lci.cs.ubbcluj.ro/ocle/

[14] Dresden OCL toolkit web site http://dresden-ocl.sourceforge.net/

[15] Coral, web site http://mde.abo.fi/tools/Coral/

[16] X. Blanc, M.-P. Gervais, P. Sriplakich, "*Model Bus : Towards the interoperability of modelling tools*", MDAFA'04, Linköping, June 10-11, 2004

[17] ArgoUML web site http://argouml.tigris.org/

[18] Pons, C., Giandini, R, Pérez., G., Pesce, P., Becker, V., Longinotti,J., Cengia,J., Kutsche, R-D., "*The PAMPERO Project: Formal Tool for the Evolutionary Software Development Process*". Home page: http://sol.info.unlp.edu.ar/eclipse. 2003

[19] Fujaba web site http://wwwcs.upb.de/cs/fujaba/index.html

PAMPERO: Precise Assistant for the Modeling Process in an Environment with Refinement Orientation

Claudia Pons[1,2], Roxana Giandini[1], Gabriela Pérez[1], Pablo Pesce[1], Valeria Becker[1], Jorge Longinotti[1], and Javier Cengia[1]

[1] LIFIA – Facultad de Informática, Universidad Nacional de La Plata, Calle 50 esq. 115. CP 1900. La Plata, Buenos Aires, Argentina
[2] CONICET (Consejo Nacional de Investigaciones Científicas y Técnicas)
cpons@info.unlp.edu.ar

1 Introduction

Abstraction [2] facilitates the understanding of complex systems by dealing with the major issues before getting involved in the detail. Apart from enabling for complexity management, the inverse of abstraction, refinement, captures the essential relationship between specification and implementation. Refinement relationship makes it possible to understand how each business goal relates to each system requirement and how each requirement relates to each facet of the design and ultimately to each line of the code. Documenting the refinement relationship between these layers allows developers to verify whether the code meets its specification or not, trace the impact of changes in the business goals and execute test assertions written in terms of abstract model's vocabulary by translating them to the concrete model's vocabulary.

Refinement has been studied in many formal notations such as Z [1] and B[4] and in different contexts, but there is still a lack of formal definitions of refinement in semi-formal languages, such as the UML. The standard modeling language UML [5] provides an artifact named *Abstraction* (a kind of Dependency) to explicitly specify abstraction/refinement relationship between UML model elements. In the UML metamodel an Abstraction is a directed relationship from a *client* (or clients) to a *supplier* (or suppliers) stating that the client (the refinement) is dependent on the supplier (the abstraction). The Abstraction artifact has a meta attribute called *mapping* designated to record the abstraction/implementation mappings, that is an explicit documentation of how the properties of an abstract element are mapped to its refined versions, and on the opposite direction, how concrete elements can be simplified to fit an abstract definition. The more formal the mapping is formulated, the more traceable across refinement steps the requirements are.

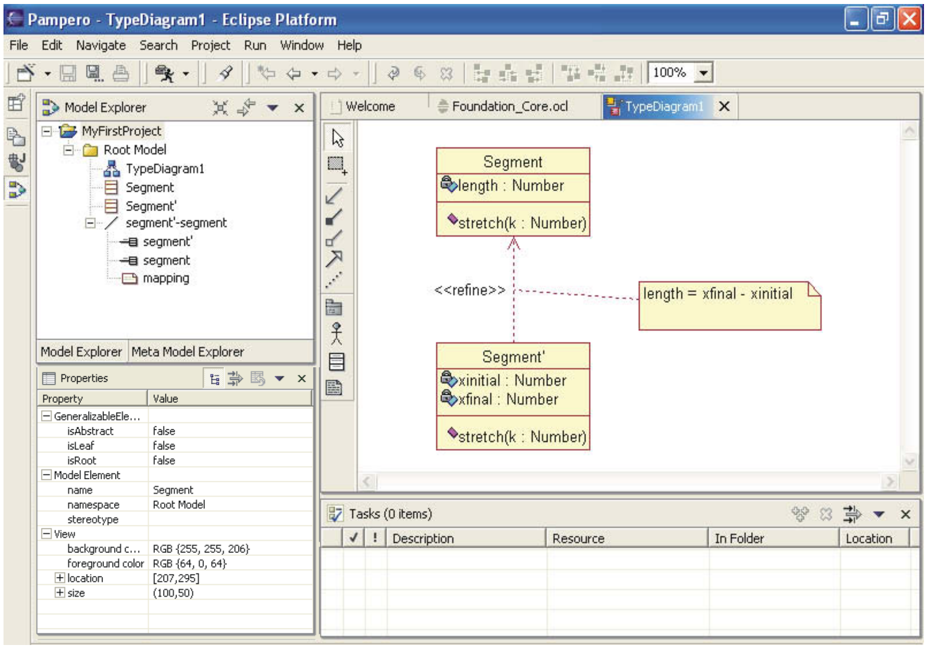
Although the Abstraction artifact allows for the explicit documentation of the abstraction/refinement relationship in UML models, an important amount of variations of abstraction/refinement remains unspecified, in general hidden under other notations. For example UML artifacts such as generalization, composite association, use case inclusion, among others, implicitly define abstraction/refinement relationship. The starting point to enable traceability across refinement steps is to discover and precisely capture the various forms of the abstraction/refinement relationship, in particular those forms which are hidden in the model.

2 Tool Support

The task of documenting refinement steps needs to be assisted by tools. To experiment, we created a tool integrated in the Eclipse environment [3], called PAMPERO (**P**recise **A**ssistant for the **M**odeling **P**rocess in an **E**nvironment with **R**efinement **O**rientation), based on the formal definition of refinement [6] [7]. The tool can be downloaded from http://sol.info.unlp.edu.ar/eclipse; it supports the documentation of explicit refinements (i.e. Abstractions artifacts with their corresponding mapping expressions) and the semi-automatic discovering and documentation of hidden refinements.

PAMPERO consists of four components: an editor, an abstraction/refinement translator, an OCL evaluator, and a detective:

The Editor. The editor supports the creation of a number of UML and OCL artifacts, including Abstractions; see figure 1. Additionally, the editor allows developers to specify the abstraction mapping attached to Abstraction artifacts, using OCL expressions.

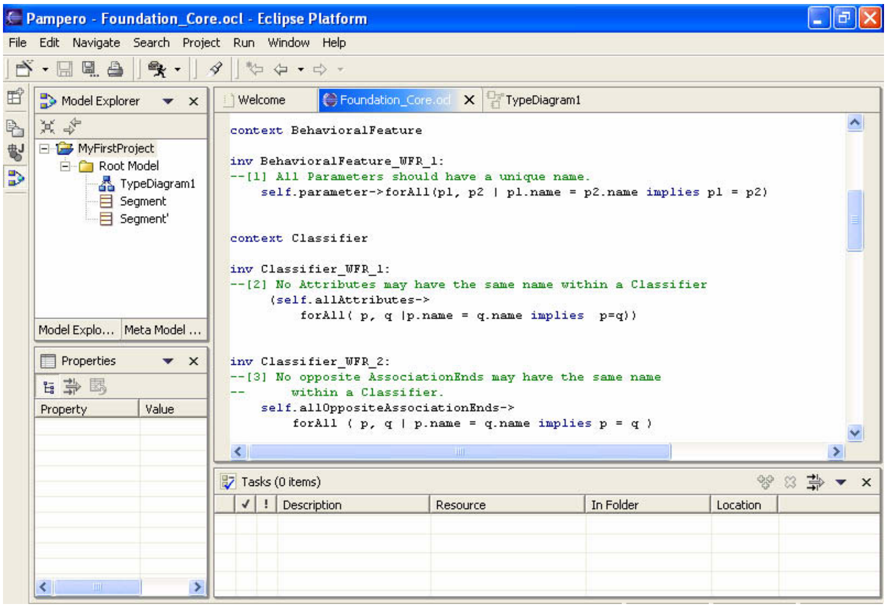


Property	Value
GeneralizableEle...	
isAbstract	false
isLeaf	false
isRoot	false
Model Element	
name	Segment
namespace	Root Model
stereotype	
View	
background c...	RGB {255, 255, 206}
foreground color	RGB {64, 0, 64}
location	[207,295]
size	(100,50)

Fig. 1. The PAMPERO tool: Edition of explicit refinement

The abstraction/refinement Translator. The translator takes an OCL expression attached to a Class and translates it to concrete vocabularies, following the refinement steps. The translation of expressions attached to elements other than Class, is not supported yet.

The evaluator. The evaluator takes OCL expressions and evaluates them on a given model. Expressions might be either originally written in the model's vocabulary or translated by the translator from another abstraction level. The evaluator was implemented following the design of the USE evaluator [8]. Figure 2 shows the evaluation of OCL well-formedness rules on the model.


Pampero - Foundation_Core.ocl - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Model Explorer
MyFirstProject
Root Model
TypeDiagram1
Segment
Segment'
Model Explo... Meta Model ...
Properties
Property Value
Welcome
Foundation_Core.ocl
TypeDiagram1
Tasks (0 items)
Description Resource In Folder Location

```
context BehavioralFeature

inv BehavioralFeature_WFR_1:
--[1] All Parameters should have a unique name.
    self.parameter->forAll(p1, p2 | p1.name = p2.name implies p1 = p2)


context Classifier

inv Classifier_WFR_1:
--[2] No Attributes may have the same name within a Classifier
     (self.allAttributes->
         forAll( p, q |p.name = q.name implies  p=q))


inv Classifier_WFR_2:
--[3] No opposite AssociationEnds may have the same name
--      within a Classifier.
    self.allOppositeAssociationEnds->
        forAll ( p, q | p.name = q.name implies p = q )
```

Fig. 2. The PAMPERO tool: Evaluation of OCL constraints

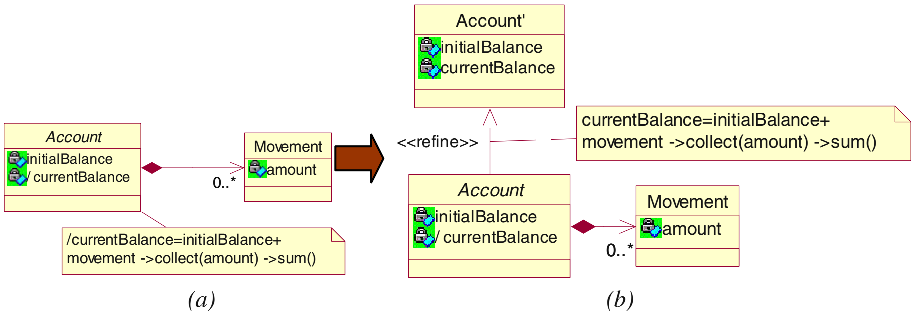


Fig. 3. Refinement hidden under decomposition: (a) Composite Association relationship. (b) Refinement relationship derived from the Composite

The Detective. This component looks into the model to discover and reveal cases of hidden refinement. The abstraction mappings automatically generated by the detective are generally in an immature state and should be completed by the developer. Figure 3 displays and example where a refinement relationship hidden under composite association is discovered and revealed by the tool. In the example the specification of the derived attribute currentBalance is suggested as mapping making it possible to translate OCL invariants such as (**Context** Account' **inv:** currentBalance>0) to a refined version such as:

Context Account inv: (initialBalance + movement->collect(amount)->sum()) > 0.

3 Conclusions

To enable traceability of requirements the presence of "undercover refinement" should be discovered and precisely documented. When the mapping between the abstract and the concrete models is explicitly (and formally) documented, assertions written in the abstract model's vocabulary can be translated, following the representation mapping, in order to analyze if they hold in the implementation. Alternatively, instances of concrete models can be abstracted according to the abstraction mapping so that abstract properties can be tested on them.

The contribution of this article is to clarify the abstraction/refinement relationship in UML models, providing basis for tools supporting the refinement driven modeling process. PAMPERO is an evidence of the feasibility of the proposal.

References

1. Derrick, J. and Boiten,E. Refinement in Z and Object-Z. Foundation and Advanced Applications. FACIT, Springer, 2001
2. Dijkstra, E.W., A Discipline of Programming. Prentice-Hall, 1976.
3. IBM, The Eclipse Project. Home Page. Copyright IBM Corp. and others, 2000-2004. http://www.eclipse.org/.
4. Lano,K. The B Language and Method. FACIT. Springer, 1996.
5. OMG. The Unified Modeling Language Specification – Version 1.5, UML Specification, revised by the OMG, http://www.omg.org, March 2003.
6. Pons, C., Pérez,G., Giandini, R., Kutsche, Ralf-D. Understanding Refinement and Specialization in the UML. 2nd International Workshop on MAnaging SPEcialization/Generalization Hierarchies (MASPEGHI). In IEEE ASE 2003, Canada.
7. Pons, C., Pérez., G. and Kutsche, R-D. Traceability across refinement steps in UML Modeling. Workshop in Software Model Engineering, 7th International Conference on the UML, October 11, 2004, Lisbon, Portugal.
8. Richters Mark and Gogolla Martin. Validating UML Models and OCL Constraints. Springer-Verlag, 2000. http://www.db.informatik.uni-remen.de/projects/USE.

Tools for Critical Systems Development with UML (Tool Demo)

Jan Jürjens* and Pasha Shabalin

Software & Systems Engineering, Dep. of Informatics, TU Munich, Germany

Abstract. The high quality development of critical systems (be it dependable, security-critical, real-time, or performance-critical systems) is difficult. Many critical systems are developed, deployed, and used that do not satisfy their criticality requirements, sometimes with spectacular failures. UML offers an opportunity for high-quality critical systems development that is feasible in an industrial context, if tools can be provided which automatically check important criticality requirements.

We present research on developing tool-support for critical systems development with UML. The developed tools can be used to check the constraints associated with UML stereotypes representing criticality requirements mechanically, based on XMI output of the diagrams from the UML drawing tool in use. We also explain a framework for implementing verification routines for the constraints associated with such stereotypes. The goal is that advanced users of the CSDUML approach should be able to use this framework to implement verification routines for the constraints of self-defined stereotypes.

Introduction. This article presents tool support for the automated analysis of UMLsec models with regard to security requirements developed at TU Munich and available at [JSA$^+$04]. It also describes a framework which allows inclusion of analysis plugins for other non-functional properties, such as dependability requirements. More details, including references to related work, can be found in [JS04, Jür04].

Background on UMLsec. We give some background on the UML extension for secure systems development called UMLsec: Recurring security requirements (such as secrecy, integrity, and authenticity) are offered as specification elements by the UMLsec extension. The properties are used to evaluate diagrams of various kinds and to indicate possible vulnerabilities. One can thus verify that the stated security requirements, if fulfilled, enforce a given security policy. One can also ensure that the requirements are actually met by the given UML specification of the system. UMLsec encapsulates knowledge on prudent security engineering and thereby makes it available to developers who may not be experts in security. The extension is given in form of a UML profile using the standard

* http://www4.in.tum.de/˜juerjens – Supported within the Verisoft Project of the German Ministry for Education and Research.

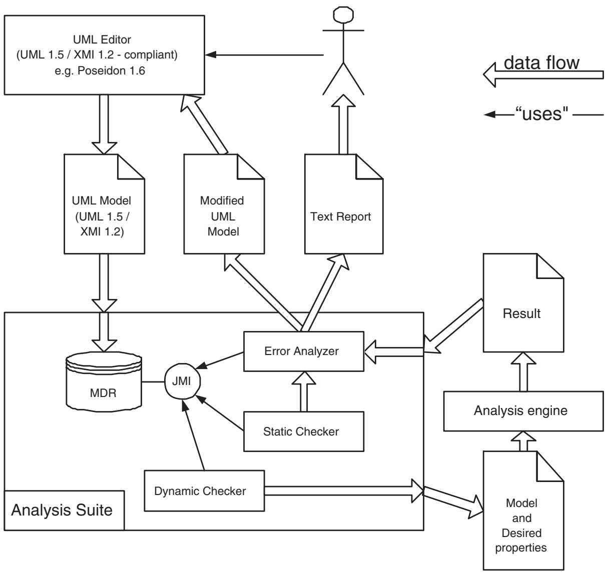


Fig. 1. UML tools suite

UML extension mechanisms. *Stereotypes* are used together with *tags* to formulate security requirements and assumptions on the system environment. *Constraints* give criteria that determine whether the requirements are met by the system design, by referring to a precise semantics mentioned below. The extension has been developed based on experiences on the model-based development of security-critical systems in industrial projects involving German government agencies and major banks, insurance companies, smart card and car manufacturers, and other companies. Note that an extension of UML to an application domain such as security-critical systems that aims to include requirements from that application domain as stereotypes, as opposed to just adding specific architectural primitives, can probably never be fully complete. We expect UMLsec to be extended with additional, more specific concepts (for example, from more specialized application domains such as mobile security).

The Framework. The architecture and basic functionality of the UMLsec analysis suite are illustrated in Fig. 1. The overall architecture is divided between the UML drawing tool in use and the analysis suite. This way the analysis suite can be offered as a web application, where the users use their drawing tools to construct the UML model which is then uploaded to the analysis suite. Additionally, a locally installable version is available. The usage of the analysis suite as illustrated in Fig. 1 proceeds as follows. The developer creates a model and stores it in the UML 1.5/XMI 1.2 file format.[1] The file is imported by the tool

[1] An upgrade to UML 2.0 is in development.

into the internal MDR repository. The tool accesses the model through the JMI interfaces generated by the MDR library. There are static checkers that parse the model, verify static features, and deliver the results to the error analyzer. Various dynamic checkers translate the relevant fragments of the UML model into the input language of several analysis engines (including model-checker and automated theorem provers). The analysis engines are spawned by the UML suite as external processes. Their results, and possibly a counter-example in case a problem was found, are delivered back to the error analyzer. The error analyzer uses the information received from both the static checkers and dynamic checkers to produce a text report for the developer describing the problems found, and a modified UML model, where the found errors are visualized and, as far as possible, corrected.

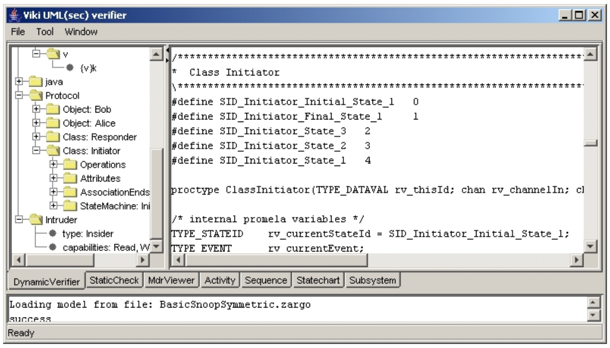
There exist various analysis plugins for the UMLsec tool framework, including:

– a tool-binding to the model-checker Spin to verify cryptographic protocols, described in [JS04],
– a tool-binding to first-order logic automated theorem provers such as e-Setheo and SPASS,
– a test-sequence generation for subsystems, sequence diagrams, activity diagrams, and statechart diagrams, and
– a checker for the static security constraints in UMLsec.

We now explain a framework for implementing verification routines for the constraints associated with the UMLsec stereotypes. The goal is that advanced users of the UMLsec approach should be able to use this framework to implement verification routines for the constraints of self-defined stereotypes. In particular, the framework includes the UMLsec tool web interface, so that new routines are also accessible over this interface. The idea behind the framework is thus to provide a common programming framework for the developers of different verification modules which in the following we just call *tools*. Thus a tool developer should be able to concentrate on the verification logic and not be required to become involved with the input/output interface. Different tools implementing verification logic modules can be independently developed and integrated. At the time of writing, there exist verification modules for most UMLsec stereotypes. An added tool implementation needs to obey the following assumptions:

– It is given a default UML model to operate 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. They receive parameters, execute, and deliver feedback.
– The tool can have an internal state which is preserved between commands.
– Each time the tool is called with a UML model, it may give back a text report and also a UML model.

These assumptions were made in order for the framework to cover as much common functionality as possible while not becoming overly complicated. Ex-



```
/*********************************************************************
*  Class Initiator
\*********************************************************************
#define SID_Initiator_Initial_State_1   0
#define SID_Initiator_Final_State_1     1
#define SID_Initiator_State_3   2
#define SID_Initiator_State_2   3
#define SID_Initiator_State_1   4

proctype ClassInitiator(TYPE_DATAVAL rv_thisId; chan rv_channelIn; cl

/* internal promela variables */
TYPE_STATEID    rv_currentStateId = SID_Initiator_Initial_State_1;
TYPE_EVENT      rv_currentEvent;
```

```
Loading model from file: BasicSnoopSymmetric.zargo
success
```

Fig. 2. UML verification framework: screenshot

perience indicates that the assumptions are not too restrictive, given the architecture in Fig. 1. The tool architecture then allows the 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 a string array as input and text as output. The framework provides default wrappers for the graphical user interface (GUI) GuiWrapper and the web mode WebWrapper. These wrappers enable use of the tool without modifications in the GUI application which is 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. The GUI is shown in Fig. 2.

Future Work. We plan to extend the framework with a functionality which allows advanced users to conveniently add self-defined stereotypes with tags and constraints to the tool-support.

References

[JS04] J. Jürjens and P. Shabalin. Automated verification of UMLsec models for security requirements. In J.-M. Jézéquel, H. Hußmann, and S. Cook, editors, *UML 2004 – The Unified Modeling Language*, volume 2460 of *LNCS*, pages 412–425. Springer, 2004.

[JSA$^+$04] J. Jürjens, P. Shabalin, E. Alter, A. Gilg, S. Höhn, D. Kopjev, M. Lehrhuber, S. Meng, M. Schwaiger, G. Kokavecz, S. Schwarzmüller, and S. Shen. UMLsec tool, 2004. Accessible through a webinterface via [Jür04]. Available as open-source.

[Jür04] J. Jürjens. UMLsec webpage, 2002-04. Accessible at http://www.umlsec.org.

[Jür04] J. Jürjens. *Secure Systems Development with UML*. Springer, 2004.

Incremental MDD Through Generative Causal Connectedness

T.D. Meijler

Univ. Of Groningen, Dept. Of Management and Organization,
Unit Information Systems
t.d.meijler@bdk.rug.nl

1 Introduction

In most MDA [4] and MDD implementations there is a separation between the modeling environment and the run-time environment. As a result, applying changes to a run-time system can be awkward. For example adding a new type to a running system means regenerating the code of the system, recompiling and restarting the system. This leads to downtime of the system, thus blocking the normal business.

Riehle et al [6] made the observation that in order to enable such changes without having to stop the run-time system and thus to allow incremental model-driven development, a direct “causal” connectedness is needed between the modeling environment and the run-time environment, such that every change or addition can also directly, and only locally (without disturbing other models and their run-time instances), affect the run-time system.

[6] describes a causally connected implementation of a UML-based model driven system. It is based on the principle of "Adaptive Object-models" (AOMs). The idea of AOMs is that an object-oriented application can be adapted through meta-data available at run-time. However, a problem of all AOM implementations known to us is that they are based on interpretation. Both instance structure and behavior is realized interpretatively [3],[7]. As a result the storage structures for instances is not efficient, e.g., each property value will be stored in a separate object. Moreover, programming support such as optimization, type checking and normal debugging is lost. The approach of [6] is therefore only used to create satisfactory applications through interactive model-driven development. The final system is still generated in a “traditional” way, and can no longer be changed incrementally.

The contribution of this short paper is that we describe how to achieve causal connectedness, and thus incremental model-driven development, using code generation. Using code generation the problems mentioned above of an interpreted realization can be circumvented, such that the causal connectedness between modeling and run-time environment can now be a fundamental, persistent feature. We have actually implemented this in our integrated modeling and execution environment. Of course due to the need for code generation with respect to [6] this is to the detriment of the possibility of interactive development.

Throughout this paper we shall use a simple example from inventory management. In inventory systems the kind of resources for which an inventory is kept cannot be defined beforehand, and changes quickly. Our example is the addition of a new kind of resource (bicycles) for which inventory must be kept.

2 Approach

We describe our approach in two main parts. The first part is the integration between modeling environment and run-time environment in order to achieve the causal connectedness between the two. This is done in order to fulfill the following prerequisites for achieving such causal connectedness:

1. In order to enable incremental changes of existing models and map these to existing (persistent) run-time instances of these models, the relationship between models and existing run-time instances must be maintained. Note however that challenges and solutions how to deal with existing instances and possible dependent models will not be treated in this paper.
2. Since the set of available models (e.g. kinds of inventory resources) is not known beforehand and varies continuously, we have to introduce explicit knowledge in the run-time environment about what this set is. We will refer to this as the need for introspection.

Thus, both a link from models to instances in the run-time environment as well as a link from the run-time environment to the available models is needed. The integration between modeling environment and run-time environment can provide this. One integrated information system (persistently) stores and presents *models* represented as objects (so-called “reified models”) as well as run-time (application) objects. Reified models are available for introspection from within the application. Note that the integration between run-time environment and modeling environment is also part of the approach described in [6].

The main question is then how to achieve integration between modeling and run-time environment such that code generation can be applied. Answering this is the second part and main contribution of our approach. First notice that in order to achieve *generative* incremental MDD there is a third major prerequisite:

3. When generating code incrementally for just one new model it must be possible to link-up the code of this new model to pre-existing realizations of pre-existing models as corresponding to the relationships between this new model and the pre-existing models, e.g., a new model specializing another model means making the new code subclass of the corresponding superclass. Thus, the relationship between models and their realization must be maintained in some way.

An integration between modeling environment and run-time environment taking code generation into account is realized through the following (“zigzag”) principle: *Each modeled object at level Mi (either a run-time object or a reified model) is defined by the model at the next level Mi+1, while technically an instance of the class generated from that model Mi+1.*

Figure 1 is used to illustrate this principle for the example. New models are shown with dashed lines. The middle column shows reified models in the form of object diagrams. As a naming convention such reified models are called “types”. Types also refer to their corresponding realization, as corresponding to 3) above. The mapping between the middle column and the right column represents the generation of classes from these types. These two columns especially illustrate our contribution. The mapping between the left column and the middle column only shows how UML models may be represented as types. The vertical axis represents the conceptual classification in M0 (object), M1 (model) and M2 (meta model).

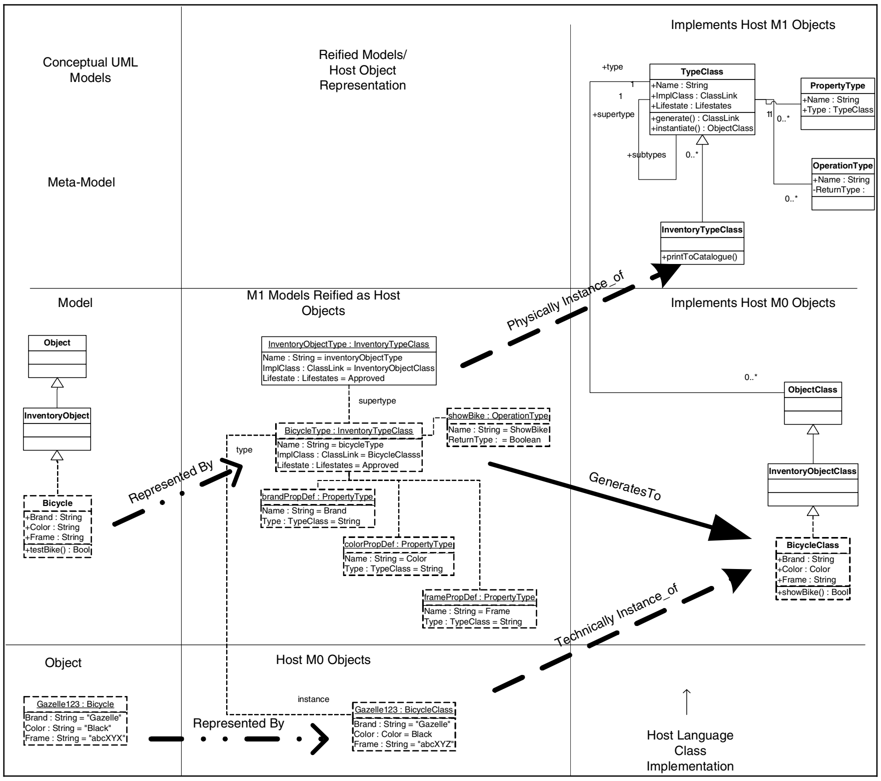


Fig. 1. Illustrating our approach for the Inventory example adding a bicycle type (dashed)

The zigzag principle –from the M0 level to the M1 level– is shown as follows: The M_0 object “Gazelle123” is modeled by the M1 type “BicycleType”. It is also a run-time (application) object instance of the class “BicycleClass” which has been generated from “BicycleType”. We thus see the integration between modeling environment and run-time environment already at this small level.

Due to this structure incremental adaptability is enabled. The reified “InventoryObjectType” model retains a relationship with its instances and a link to its realization (realizing points 1 and 3 above). When for the new type “BicycleType” code has been generated and compiled, the type is run-time available to be instantiated and to be queried for introspection (realizing point 2 above). Users of other types and their instances need not be bothered.

3 Other Related Work

Our approach is similar to the approach taken in the Smalltalk Virtual Machine [4] with its metalevel architecture reifying classes as objects and its mixing between interpretation and compilation. In our case models are reified offering further abstraction, allowing generating to different technical platforms as usual in MDA [4].

Causal connectedness has already been realized (although not explicitly named as such) in other areas, for example Database Management Systems [1]. In such systems incremental changes on tables are possible and have a direct effect on the data. It is also common use to store meta-data (a catalog or dictionary) as data, similar as what we do with reified models.

4 Consequences and Further Work

In this paper we describe only a small part. Causal connectedness enables an explicit representation of relationships between all meta-levels M0, M1, M2 and M3. We focus on the link between M0 and M1 to illustrate our approach. Moreover, while causal connectedness is meant to support full incremental adaptability, we don’t discuss how to handle the complications of the adaptation of existing models, with their possible existing instances and existing dependent models. Also, we don’t give a formalized underpinning; see for that earlier work of the author [5]. We don’t discuss the full mapping to MDA, and give only minimal details of our realization, this will also be subject for further publications.

The following consequences of this approach are also subject of further work. The approach integrates the modeling environment with the run-time environment, thus promising such features as learning, and model-driven ad-hoc intervention in executing objects. It moreover promises extensibility of the modeling environment for new domain specific modeling constructs, based on the same incremental step from meta-modeling in M2 to implementation classes for M1 objects.

Through class generation, we overcome the disadvantage of the interpreted realization of Causal Connectedness on the basis of Adaptive Object Models [6]. We believe that this technique therefore makes causal connectedness mature and a serious candidate for realizing applications as we are actually doing. This work has been realized in an experimental development environment for model-driven development and execution of enterprise systems called Nucleus.

References

[1] Date, C.J., An Introduction to Database Systems. Addison Wesley 2000

[2] Goldberg, A., Robson, D., Smalltalk-80: The Language and Its Implementation, Addison-Wesley, ISBN 0-201-11371-6, 1983

[3] Johnson, R., Wolf, B. "Type Object". In Pattern Languages of Program Design 3. Addison Wesley, 1998

[4] Kleppe, A., Warmer, J., Bast, W., MDA Explained, Addison-Wesley, ISBN 0-321-19442-X

[5] Meijler, T.D., User-level Integration of Data and Operation Resources by means of a Self-Descriptive Data Model, Ph.D. Thesis Part II Formalization, Erasmus University Rotterdam, 1993

[6] Riehle, D., Fraleigh S., Bucka-Lassen, D., and Omorgbe, N. 2001. "The architecture of a UML virtual machine". In Proceedings of OOPSLA'01. ACM Press, New York, 327–341

[7] Yoder, J.W., Balaguer, F., and Johnson, R. 2001. "Architecture and Design of Adaptive Object-Models". In ACM SIGPLAN Notices, 36(12), December 2001

Model-Driven Engineering of Middleware-Mediated Distributed Systems

Raul Silaghi and Alfred Strohmeier

Software Engineering Laboratory,
Swiss Federal Institute of Technology in Lausanne,
CH-1015 Lausanne EPFL, Switzerland
{Raul.Silaghi, Alfred.Strohmeier}@epfl.ch

Abstract. Existing software engineering methods tend to have a strong focus on functional requirements, ignoring more or less non-functional concerns, such as middleware-specific concerns, which have to be addressed sooner or later when designing and implementing distributed systems. Following an MDA approach to software development, the Enterprise Fondue method proposes a hierarchy of UML profiles as a means for addressing middleware-specific concerns at different MDA-levels of abstraction, along with model transformations to incrementally refine existing design models according to the proposed profiles. Tool support is provided through the Parallax framework, which assists developers in the Enterprise Fondue refinement process and enables them to modularize middleware-specific crosscutting concerns into aspect-promoting Eclipse plug-ins.

Keywords: MDA, UML Profiles, Model Transformations, Middleware Concerns, Enterprise Fondue, Parallax, Eclipse Plug-ins, AOP, AspectJ.

1 Introduction

Over the last decade, middleware has become an integral part in the development of distributed enterprise systems. Distributed technologies, such as COM/DCOM/COM+, RMI, CCM/CORBA, Jini, EJB/J2EE, .NET, and Web Services, are commonly referred to as middleware and have been increasingly adopted by many enterprises as the backbone of their IT infrastructure. Besides the object, component, or service middleware technologies mentioned above, other types of middleware take different approaches, such as distributed transactions, message passing, or remote procedure calls [1].

In this plethora of middleware platforms it is often hard to identify the right one for designing and implementing a given distributed system on top of it [2]. Moreover, a problem often encountered in software projects in general, and distributed systems in particular, is that the development of business logic is dominated by technical details that do not really contribute to the functionality of the software system but hinder nevertheless the analysis and design of the business logic. A recognized challenge for software engineering research is to devise notations, techniques, methods, and tools for distributed system development that systematically build and exploit the capabilities that middleware deliver [1].

To escape from the proliferation of middleware infrastructures and to avoid drowning in their implementation complexities, *models* are proposed as a far more accessible and easier means for developers to build, extend, and evaluate applications than working directly at the code level. The Model Driven Architecture (MDA) [3], an Object Management Group initiative, promotes the separation of concerns between two modeling dimensions: one focusing on the business functionality (resulting in *Platform Independent Models – PIMs*), and the other one focusing on the implementation of that functionality on a specific middleware platform (resulting in *Platform Specific Models – PSMs*). While model transformations should be used to refine PIMs into PSMs, code generators are supposed to map PSMs to concrete middleware-based implementations, providing thus an elegant approach to adapt PIMs to the peculiarities of the new middleware infrastructures that do not cease to appear.

Before going any further, referring to the "myth of absolute platform independence" and "platform relativism" [4], and in order not to leave any doubts or to risk any misinterpretations, we would like to make clear that, in the context of this work, we consider the *middleware* to be our MDA platform, and not the operating system, or anything else. Moreover, even though MDA is completely independent of any modeling language, the Unified Modeling Language (UML) [5] established itself as the de-facto standard. As a consequence, we only focus on the UML support for MDA.

For the MDA approach to software development to become a reality for distributed enterprise systems, MDA needs to provide support for understanding, describing, and implementing different middleware-specific concerns, such as distribution, concurrency, transactions, security, etc., also referred to as *pervasive services* in MDA's PIM terminology [6]. However, the current UML [5] does not provide any specific or standard support for modeling pervasive services. What it does offer, is the possibility to "extend" the UML metamodel through, and only through, *profiling*, which defines how specific UML model elements are customized and extended with new semantics as if they were instances of new "virtual" metamodel constructs. This unique position of UML profiles makes them play a key role in MDA, since developers must know about, or define, the metamodels of their input and output models before implementing any model transformation.

2 Enterprise Fondue

The MDA-compliant Enterprise Fondue [7] software development method defines MDA-oriented UML profiles that address middleware-specific concerns at different levels of abstraction. It also promotes a systematic approach to addressing pervasive services in an MDA-compliant manner, at different levels of abstraction, through incremental refinement steps along middleware-specific concern-dimensions according to the proposed UML profiles.

A complete example has already been carried out for the distribution concern. The hierarchy of *UML-D Profiles* proposed in [8] addresses the distribution concern in an MDA-oriented fashion at three different levels of abstraction: *platform-independent* level, *abstract realization* level, and *concrete realization* level. MTL [9] model transformations were proposed to refine existing design models (within the same or between different MDA-levels) along distribution-related concern-dimensions and in conformance with the UML-D Profiles. The CORBA technology was used to illustrate how the refinement process is applied to a concrete example.

Relying on the PIM-level outcome of the refinement along the distribution concern-dimension as presented in [8], we showed in [10] how concurrency induced by distribution can be inferred in an automatic way, provided that a small set of JavaBeans-like design conventions are strictly adhered to. A simple PIM-level concurrency profile was considered in order to illustrate how the inference algorithm evolves on a concrete example and how an initial distributed design is automatically refined along the concurrency concern-dimension according to the proposed concurrency profile.

3 Parallax

In order to proliferate the MDA vision and make it a reality, and at the same time to facilitate the development of distributed middleware-mediated applications, there is an imperative need for tool support. Unfortunately, so far there is very little in terms of concrete tools that actually support MDA beyond traditional UML modeling and skeleton-class generation.

Based on the solid foundations of the MDA-oriented UML profiles defined in the context of the Enterprise Fondue method, and in order to provide developers with integrated tool support that allows them to incrementally apply these profiles for refining their design models along middleware-specific concern-dimensions at different stages in the development life cycle of distributed enterprise applications, we designed the *Parallax* framework [11].

Implemented as an Eclipse plug-in, Parallax relies on a well-defined system of plug-ins and on aspect-oriented support (AspectJ [12]) to address middleware crosscutting concerns at different MDA-levels of abstraction. Parallax supports the MDA approach to software development by enabling developers to refine their designs along middleware-specific concern-dimensions, and to view their enhanced designs through a prism of middleware platforms and see how middleware concerns are actually implemented at the code level [13].

4 Future Work

Both the Enterprise Fondue method and the Parallax tool support are relatively young, and still undergoing refinement and improvement as we move along. Nevertheless, they are both applied to case studies and tests are carried out to determine their limitations and extensibility problems, and to adjust them accordingly.

With regard to the Enterprise Fondue method, further investigations will be carried out to check whether other middleware-specific concerns lend themselves to such an MDA-oriented profiling approach. Addressing concurrency (*UML-C Profiles*), transactions (*UML-T Profiles*), security (*UML-S Profiles*), global time (*UML-GT Profiles*), etc., will be intermediate steps towards an MDA-Oriented UML Profile for Middleware Services, or more precisely Middleware-Specific Concerns (*UML-MS Profiles*).

Regarding Parallax, once the extension points that we have been experimenting with become stable, we intend to follow the Eclipse contribution circle and publish them, so that other developers and middleware vendors may contribute and enrich Parallax by implementing and providing the community with new Parallax plug-ins addressing middleware-specific concerns for their favorite middleware infrastructures.

In order to increase the modularization of middleware crosscutting concerns in the final application as well, we will explore the possibility of *generating aspects* for the targeted programming language, such as AspectJ, AspectC++, to encapsulate such concerns.

References

[1] Emmerich, W.: *Software Engineering and Middleware: A Roadmap*. Proceedings of the Future of Software Engineering Track, FoSE, held at the 22nd International Conference on Software Engineering, ICSE, Limerick Ireland, June 4-11, 2000. ACM Press, 2000, pp. 117 – 129.

[2] Zarras, A.: *A Comparison Framework for Middleware Infrastructures*. Journal of Object Technology, **3**(5), May-June 2004, pp. 103 – 123.

[3] Object Management Group, Inc.: *Model Driven Architecture*. `http://www.omg.org/mda/`, November 2004.

[4] Frankel, D. S.: *The MDA Marketing Message and the MDA Reality*. MDA Journal, a Business Process Trends Column, March 2004. `http://www.bptrends.com/`.

[5] Object Management Group, Inc.: *Unified Modeling Language Superstructure Specification*, v2.0, August 2003.

[6] Miller, J.; Mukerji, J.: *Model Driven Architecture (MDA)*. Object Management Group, Document `ormsc/2001-07-01`, July 2001.

[7] Silaghi, R.; Strohmeier, A.: *Integrating CBSE, SoC, MDA, and AOP in a Software Development Method*. Proceedings of the 7th IEEE International Enterprise Distributed Object Computing Conference, EDOC, Brisbane, Queensland, Australia, September 16-19, 2003. IEEE Computer Society, 2003, pp. 136 – 146. Also available as Technical Report, N° IC/2003/57, Swiss Federal Institute of Technology in Lausanne, Switzerland, September 2003.

[8] Silaghi, R.; Fondement, F.; Strohmeier, A.: *Towards an MDA-Oriented UML Profile for Distribution*. Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference, EDOC, Monterey, CA, USA, September 20-24, 2004. IEEE Computer Society, 2004, pp. 227 – 239. Also available as Technical Report, N° IC/2004/49, Swiss Federal Institute of Technology in Lausanne, Switzerland, May 2004.

[9] French National Institute for Research in Computer Science and Control (INRIA): *Model Transformation Language (MTL)*. `http://modelware.inria.fr/`, August 2004.

[10] Silaghi, R.; Strohmeier, A.: *An MDA-Based Approach for Inferring Concurrency in Distributed Systems*. Proceedings of the 4th International Workshop on scientiFic engIneering of Distributed Java applIcations, FIDJI, Luxembourg-Kirchberg, Luxembourg, November 24-25, 2004. Springer-Verlag, Lecture Notes in Computer Science, 2005 (to appear). Also available as Technical Report, N° IC/2004/83, Swiss Federal Institute of Technology in Lausanne, Switzerland, November 2004.

[11] Software Engineering Laboratory at the Swiss Federal Institute of Technology in Lausanne: *The Parallax Project*. `http://parallax-lgl.epfl.ch/`, November 2004.

[12] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. G.: *An Overview of AspectJ*. Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP, Budapest, Hungary, June 18-22, 2001. LNCS Vol. **2072**, Springer-Verlag, 2001, pp. 327 – 353.

[13] Silaghi, R., Strohmeier, A.: *Parallax, or Viewing Designs Through a Prism of Middleware Platforms*. Proceedings of the 38th Annual Hawaii International Conference on System Sciences, HICSS, Hilton Waikoloa Village, Big Island of Hawaii, HI, USA, January 3-6, 2005, part of the Mini-track on Adaptive and Evolvable Software Systems, AESS. IEEE Computer Society, 2005. Also available as Technical Report, N° IC/2004/69, Swiss Federal Institute of Technology in Lausanne, Switzerland, August 2004.

Profile Suite for Model Transformations on the Computation Independent Level

Michał Śmiałek

Warsaw University of Technology and Infovide S.A.,
Warsaw, Poland
smialek@iem.pw.edu.pl, http://www.iem.pw.edu.pl/~smialek/

Abstract. Transformations of models on the Computation Independent level of the MDA framework seem to have little focus in the current research. This is despite their great significance for ensuring traceability of business requirements into system design and implementation. Hence, this paper proposes a suite of profiles that define appropriate models and transformations between them. There is briefly described a set of well-formedness rules for the models, and a set of mappings and rules that allow for model transformations. These mappings and rules are presented in the context of UML 2.0 Profiles.

1 Introduction and Rationale

Modern businesses change their processes rapidly and require rapid changes of their supporting software systems. This trend enforces the organizations to use model-based notations to describe their businesses in order to overcome complexity. MDA [7] with its idea of precise automatic model transformations significantly supports this tendency [5]. Although widely used, model-based notations seem to lack in precise and standard metamodels to define contents of individual requirement chunks. These chunks should be written with a notation that is simple enough to be easily understood by the users and to provide for unambiguous traceability into design models, allowing for different views of the prospective system [3].

Therefore, while most of the current research concentrates on transformations between the design level models (PIM and PSM) [2], we shall describe transformations on the computation independent level (CIM). In this paper we propose a set of profiles that define notation for individual use cases and relate them to the business domain model. We also complete the mapping with appropriate transformation rules [1].

2 Profiles for the Computation Independent Level

We can describe any software development effort as a composition of notation, techniques and technical process [6]. UML-based methodologies use Profiles [8] to define well-formedness rules for elements of their notation. For the computation independent level we thus propose three profiles: Business, Functional and Domain (Fig. 2-3). In order for these models to maintain coherency, we will extend the area where Profiles are used by defining transformation techniques with Transformation Profiles (Fig. 1).

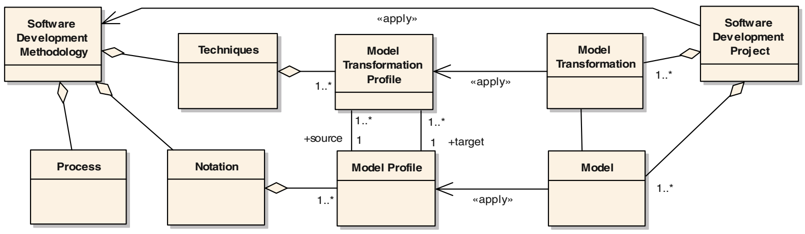


Fig. 1. Model profiles and transformation profiles shown as components of a methodology

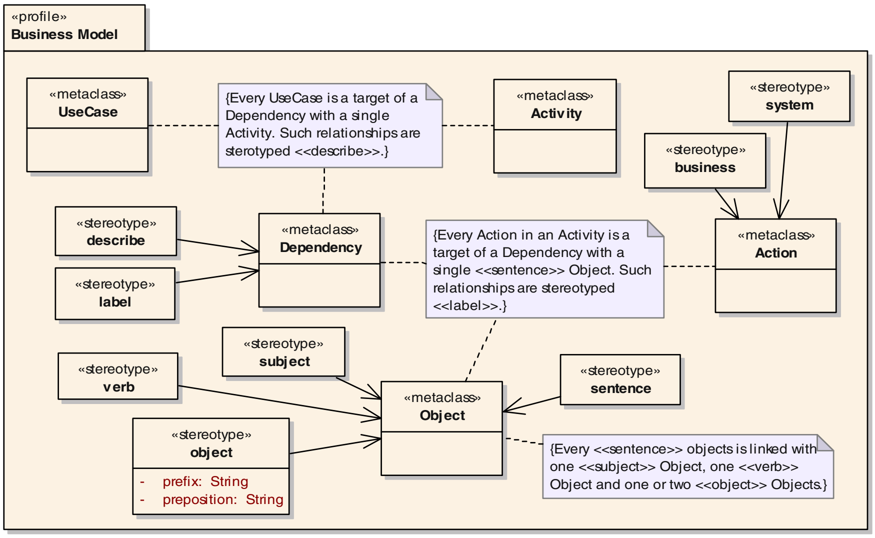

«profile» Business Model
«metaclass» UseCase
{Every UseCase is a target of a Dependency with a single Activity. Such relationships are sterotyped <<describe>>.}
«metaclass» Activity
«stereotype» system
«stereotype» business
«stereotype» describe
«stereotype» label
«metaclass» Dependency
{Every Action in an Activity is a target of a Dependency with a single <<sentence>> Object. Such relationships are stereotyped <<label>>.}
«metaclass» Action
«stereotype» subject
«stereotype» verb
«metaclass» Object
«stereotype» sentence
«stereotype» object
- prefix: String
- preposition: String
{Every <<sentence>> objects is linked with one <<subject>> Object, one <<verb>> Object and one or two <<object>> Objects.}

Fig. 2. Overview of the Business Model Profile

Business and Functional Model Profiles. The models are composed of UseCases and Activities [8] (Fig. 2). We put a constraint that every UseCase is «described» with a single Activity. Activities contain Actions (among other possible elements), that are «labeled» with «sentences». Every sentence is an Object linked with three or four other Objects («subject», «verb» and one or two «objects» - SVO[O] sentence). The business profile also assigns stereotypes to Actions («system» and «business»). These stereotypes determine Action transformation into appropriate elements of other models. We also allow for the sentences to be written textually in order to form scenarios.

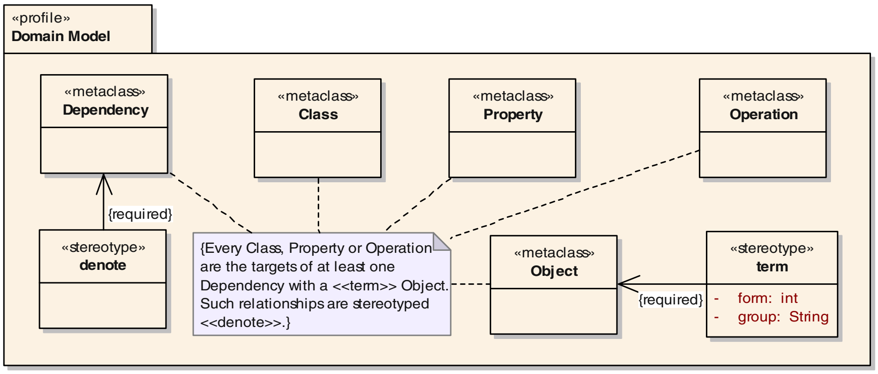


Fig. 3. Overview of the Domain Model Profile

Profile for the Domain Model. The domain model is basically composed of Classes with Properties and Operations (Fig. 3) that represent notions in the problem domain. These notions can be «denoted» with «terms». These «term» Objects allow for various synonyms and forms of the basic notion. The synonyms can be grouped for different groups of users (see the attributes of the «term» stereotype).

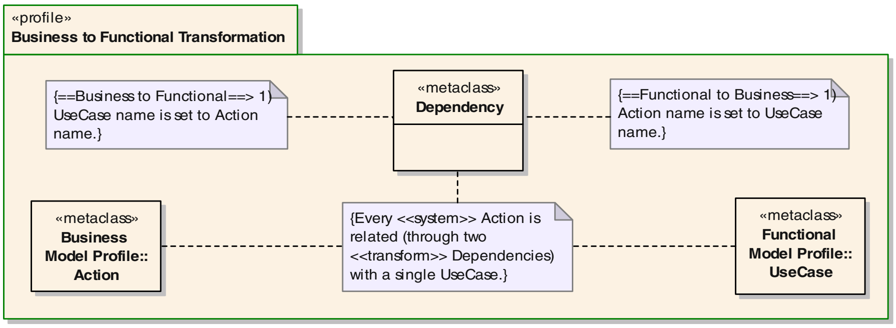

«profile»
Business to Functional Transformation
{==Business to Functional==> 1) UseCase name is set to Action name.}
«metaclass» Dependency
{==Functional to Business==> 1) Action name is set to UseCase name.}
«metaclass» Business Model Profile:: Action
{Every <<system>> Action is related (through two <<transform>> Dependencies) with a single UseCase.}
«metaclass» Functional Model Profile:: UseCase

Fig. 4. Overview of the Business to Functional Transformation Profile

Business to Functional Transformation Profile. The transformation maps «system» Actions to functional UseCases (Fig. 4). The transformation is straightforward and simply synchronizes appropriate names.

Business and Functional to Domain Transformation Profile. This transformation maps «verb» and «object» Objects to appropriate Classes, Properties or Operations (Fig. 5). The Objects that are not yet mapped have to be mapped manually to an element of the domain model. Their names are mapped to one of the dependent «terms». In the case where no appropriate notion or term exists, a new element has to be added to the domain model. Transformation in the reverse direction depends on the user group chosen. For each of the notions, the group determines the «term» that is to be used to set the names of the dependent «verbs» or «objects» in the business model.

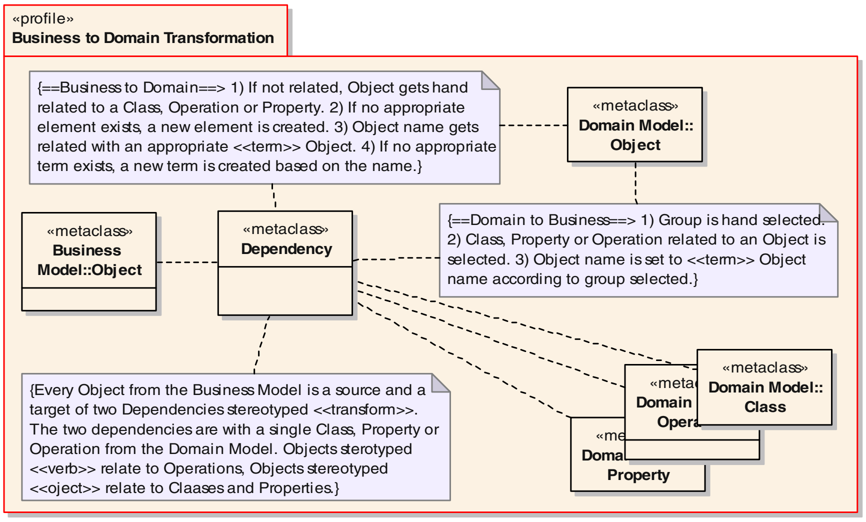

«profile»
Business to Domain Transformation
{==Business to Domain==> 1) If not related, Object gets hand related to a Class, Operation or Property. 2) If no appropriate element exists, a new element is created. 3) Object name gets related with an appropriate <<term>> Object. 4) If no appropriate term exists, a new term is created based on the name.}
«metaclass» Domain Model:: Object
«metaclass» Business Model::Object
«metaclass» Dependency
{==Domain to Business==> 1) Group is hand selected. 2) Class, Property or Operation related to an Object is selected. 3) Object name is set to <<term>> Object name according to group selected.}
«metaclass» Domain Model:: Class
«meta Domain Opera
«me Doma Property
{Every Object from the Business Model is a source and a target of two Dependencies stereotyped <<transform>>. The two dependencies are with a single Class, Property or Operation from the Domain Model. Objects sterotyped <<verb>> relate to Operations, Objects stereotyped <<oject>> relate to Claases and Properties.}

Fig. 5. Overview of the Business to Domain Transformation Profile

3 Conclusions and Future Work

The presented profiles were verified in several commercial projects that include re-engineering of business processes with underlying software systems in an IT department of a major Polish telecom and business process and systems specification for a large emerging economic information bureau. Application of the profiles allowed for precise synchronization between behavioral and structural models on the CI level. Experience showed that keeping models synchronized gave a very significant positive shift to business-IT alignment. It could be noticed that the transformation profiles allowed for almost instantaneous propagation of changes from the business model to the requirement specifications of the related software systems. This was even though all the transformations were performed by hand in a standard CASE tool.

Tool support was very desired by the analysts that had to synchronize models by hand and the future work shall be concentrated on constructing of such a tool. The new tool should obviously allow for automatic verification of model well-formedness and execution of model transformations. It can be noted that the tool can be based on the existing CASE tools (the repository metamodel is not changed by the Profile mechanism). Sentences, terms, notions and other elements of the proposed profiles can thus be stored in the existing repository structures. Only a plug-in is needed to simplify and automate the manipulation of models, notably for writing SVO[O] sentences aligned with the domain vocabulary of terms (see [4] for an example).

References

1. Bezivin, J., Gerbe, O.: Towards a Precise Definition of the OMG/MDA Framework. Proc. 16th Int. Conf. on Automated Software Engineering (2001) 273-280
2. Czarnecki, K., Hensen S.: Classification of Model Transformation Approaches. OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture (2003)
3. Dijkman, R.M., Quartel, D.A.C., Pires, L.F., van Sinderen, M.J.: An Approach to Relate Viewpoints and Modeling Languages. Proc. 7th Int. EDOC Conference (2003) 14-27
4. Gryczon, P., Stańczuk, P.: Obiektowy system konstrukcji scenariuszy przypadków użycia (Object-Oriented Use Case Scenario Construction System). MSc thesis, Warsaw University of Technology (2002) http://www.iem.pw.edu.pl/~smialek/mgr/PGryczon_PStanczuk.pdf
5. Harmon, P.: The OMG's Model Driven Architecture and BPM. Business Process Trends Newsletter, vol. 2, no. 5 (2004) 1-11
6. Henderson-Sellers, B., Simons, A.J.H., Younessi, H.: The OPEN Toolbox of Techniques. Addison-Wesley Longman (1998)
7. MDA Guide Version 1.0.1. omg/2003-06-01, OMG (2003)
8. Unified Modeling Language: Superstructure. ptc/03-08-02, OMG (2003)

The ProjectIT-RSL Language Overview

Carlos Videira[1], João Leonardo Carmo[2], and Alberto Rodrigues da Silva[3]

[1] INESC-ID and Universidade Autónoma de Lisboa,
Rua de Santa Marta, nº 56, 1169-023 Lisboa, Portugal
cvideira@acm.org
[2] INESC-ID and Instituto Superior Técnico,
Rua Alves Redol, nº 9 –1000-029 Lisboa, Portugal
joao_leonardo@netcabo.pt
[3] INESC-ID and Instituto Superior Técnico,
Rua Alves Redol, nº 9 –1000-029 Lisboa, Portugal
alberto.silva@acm.org

Abstract. Requirements engineering is widely considered to be an essential activity for the successful development of information systems. This paper briefly presents a new initiative called "ProjectIT-Requirements" and describes the results achieved in the definition of a requirements specification language, called "ProjectIT-RSL", and the implementation of a prototype using VisualStudio.NET. This is the first step of a process that will enable the automatic generation of UML models and programming code, based on the MDD approach.

1 Introduction

In the Information Systems Group of INESC-ID [http://gsi.inesc-id.pt/] we have been developing efforts to increase the productivity of the software development process. Recently we started a research program, called "ProjectIT", whose goal is to develop a complete software development workbench with support for requirements engineering, analysis and design, generative programming techniques, and project management activities [5]. Our experience in the area of requirements engineering led us to conclude that to achieve success, requirements must somehow be formalized, and although previous initiatives were an important contribution [3], for a number of different reasons they have not been widely adopted. On the other hand, most of the existing tools are above all requirements management tools that do not automate important tasks and reduce the work involved in the overall process. For example, they do not promote the reuse of requirements, neither do they allow a tight integration between requirements and other software process artefacts (such as models and code).

Within this context we started the ProjectIT-Requirements project [6], which combines the benefits of formalizing the requirements specification with the need to use a simple notation understandable by non-technical stakeholders. Our hypothesis is that we should adopt a "controlled natural language", a subset of natural language with specific rules for requirements specification, with a limited vocabulary and a simplified grammar.

2 The ProjectIT-RSL Language

After defining the overall ProjectIT-Requirements architecture [6], we developed an initial prototype of the main deliverables of the project, upon which all the others depend: the **Requirements Language**, defined by a metamodel (a brief overview is shown in Figure 1), and by a grammar for defining the rules to map these concepts into sentences; the **Requirements Editor**, which acts like a traditional editor for introducing controlled requirements text; and the **Requirements Compiler** and **Intellisense** features, responsible for checking the requirements definitions, introduced in the editor, against the syntactic and semantic rules defined by language.

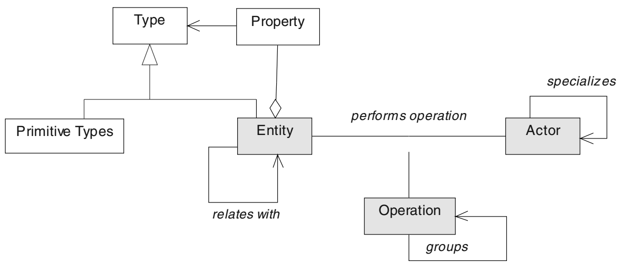


Fig. 1. ProjectIT-RSL metamodel

Our initial proposal for the ProjectIT-RSL language is based upon the analysis of how requirements are most often described: normal language sentences where *actors* carry out *actions* which imply the access to one or more *entities*.

- **Actors** are defined as active resources (e.g., an external system or an end-user) that perform operations involving typically one or more entities.
- **Entities** are the static resources that are affected by the operations (e.g., a document or the data about a client or an invoice stored in a database). Entities have **Properties** that represent and describe their state.
- **Operations** are described by their respective workflows, which consist of a sequence of simpler Operations that affect Entities. This recursive definition will end in atomic and primitive Operations (e.g., create, update or delete operations) provided by default by our framework.

3 Prototype Development

In order to evaluate the preliminary version of the ProjectIT-RSL in a high-productive environment we decided to take the benefits of the features provided by Visual Studio .NET and .NET Framework [http://msdn.microsoft.com/] and so we decided to build a prototype in this environment. The choice is quite obvious, since this development environment provides some of the features we consider important such as intellisense and syntax validation when writing code. Microsoft also provides the Visual Studio

Industry Partner Program [http://msdn.microsoft.com/vstudio/extend/], abbreviated VSIP, which are a set of COM APIs that enable the integration of new features in the Visual Studio.NET development environment, such as the possibility of adding new languages, or creating new types of projects.

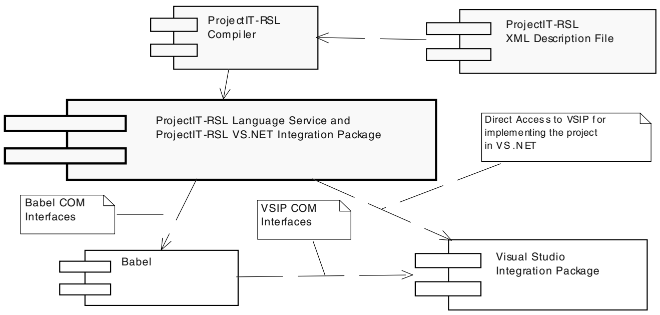
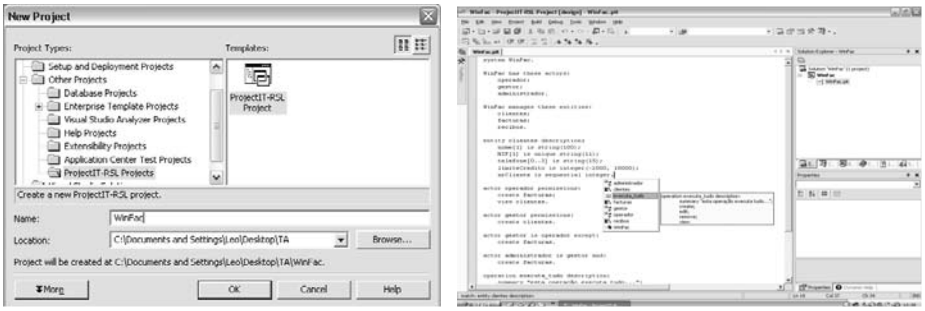

ProjectIT-RSL Compiler
ProjectIT-RSL XML Description File
ProjectIT-RSL Language Service and ProjectIT-RSL VS.NET Integration Package
Direct Access to VSIP for implementing the project in VS.NET
Babel COM Interfaces
VSIP COM Interfaces
Babel
Visual Studio Integration Package

Fig. 2. ProjectIT-Requirements tools architecture

These VSIP APIs are very powerful, but difficult to understand and very low-level from the developer's point of view (they are written in C++ and the code is difficult to understand). Hence, we decided to use the Babel library that comes together with VSIP, but provides a higher abstraction layer above it. For example, the intellisense feature is handled directly by Babel and can be accessed from it. We also decided to use implementations of Lex and Yacc (Flex [http://www.gnu.org/software/flex/] and Bison [http://www.gnu.org/software/bison/bison.html]) to implement the Language Checker in the Editor, and also for the compiler of the ProjectIT-RSL, based upon syntax and semantic rules well known from programming languages. Figure 2 shows how the different components of the architecture integrate to provide the global functionality.



Create a new ProjectIT-RSL project.

Name: WinFac

Location: C:\Documents and Settings\Leo\Desktop\TA

Project will be created at C:\Documents and Settings\Leo\Desktop\TA\WinFac.

```
system WinFac;

WinFac has these actors:
    operador;
    gestor;
    administrador;

WinFac manages these entities:
    clientes;
    facturas;
    recibos.

entity clientes description:
    nome[1] is string(100);
    NIF[1] is unique string(11);
    telefone[0..3] is string(15);
    limiteCredito is integer(-1000, 10000);
    noCliente is sequential integer;

actor operador permissions:
    create facturas;
    view clientes.

actor gestor permissions:
    create clientes.

actor gestor is operador except:
    create facturas.

actor administrador is gestor and:
    create facturas.

operation executa_tudo description:
    summary "esta operação executa tudo...";
```

Fig. 3. Creating a new ProjectIT-RSL project and using the ProjectIT-RSL editor

We have already implemented a complete prototype integrated in Visual Studio.NET, which includes: (1) the possibility of creating new projects (as shown in

Figure 3); (2) the ProjectIT–RSL editor (also in Figure 3), which supports full syntax highlighting, the auto-complete feature and on-the-fly syntactic language verification, suggesting the available correction choices; and (3) the ProjectIT-RSL compiler.

Besides detecting additional errors in the requirements written, the compiler produces a XML file currently conformant with the XIS/UML profile [4], which will be the input to ProjectIT-MDD, the component of ProjectIT that is intended to specify, simulate and develop information systems according with the MDD approach. In this respect, our work has some similarities with the projects described in [1] and [2], although we will follow a less formal approach [6].

4 Conclusions and Future Work

The results achieved with this prototype and with the current ProjectIT-RSL metamodel confirm that our proposal is a suitable and effective approach to requirements specification. Using a controlled natural language to help requirements description will ease the involvement of the non-technical stakeholders in the requirements specification and management process.

The future work, besides improving ProjectIT-RSL (for example, adding support for the specification of more complex and versatile operations and workflows), includes the integration of these features in a non-proprietary development environment (the ProjectIT-Studio tool [4]). Simultaneously, we will research for more advanced features, such as requirements reuse based on requirements architectures.

Although the ProjectIT-RSL is an initiative in the area of requirements specification, the evolution of the ProjectIT-RSL language will integrate and extend the current XIS/UML profile [4], resulting in the ProjectIT/UML profile, a common metamodel for all ProjectIT sub-systems [5]. This profile will provide mechanisms to support the automatic generation of UML models of the system to be developed, and when processed by ProjectIT-MDD tools, will enable the generation of design and code artefacts. This will be a seamless integration, supported by the capability of ProjectIT-MDD tools to read the XML generated by ProjectIT-Requirements.

References

1. Bryant, B., et al. (2003) From Natural Language Requirements to Executable Models of Software Components, Proc. of Monterey Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation, pp. 51-58, Chicago, Illinois
2. Lamsweerde, A. (2003) Goal Oriented Requirements Engineering: From System Objectives to UML Models to Precise Software Specifications, Proc. of the 25th International Conference on Software Engineeering, IEEE Computer Society.
3. Lamsweerde, A. (2000) Formal Specification: a Roadmap, Proceedings of the conference on The future of Software engineering, pp 147-159, Limerick, Ireland.
4. Silva, A. R., Lemos, G., Matias, T., Costa, M. (2003) The XIS Generative Programming Techniques, Proc.of the 27th COMPSAC Conference, IEEE Computer Society.
5. Silva, A. R. (2004) O Programa de Investigação “ProjectIT”, White Paper, Relatório INESC-ID, Grupo de Sistemas de Informação.
6. Videira, C., Silva, A. R. (2004) ProjectIT-Requirements, a Formal and User-oriented Approach to Requirements Specification, accepted for publication in the JIISIC 2004 Conference.

A UML-Based Tool for Designing User Interfaces

Pedro F. Campos and Nuno J. Nunes

University of Madeira – Dep. of Mathematics and Engineering,
Campus Universitário da Penteada,
9000-930 Funchal – Portugal

Abstract. Existing software modeling tools are widely recognized to be hard to use and, hence, to adopt. We believe those usability problems are related to a legacy of formalism-centric tools that don't promote the new challenges of modern software development. In this short paper, we briefly describe a new tool, under development, that tries to promote usability in modeling tools to support collaborative development of interactive software. It focuses on usable, real-world languages and a developer-centered design.

1 Introduction

Many different approaches have been proposed to capture the presentation aspects of interactive systems [5]. These include sketches, content inventories, wire-frame schematics and class stereotypes. UML class stereotypes have become a very popular alternative to structure the presentation aspects of interactive systems. Popular object-oriented methods extend the concept of a class to convey the important presentation elements, those concepts are know as "views" in Ovid, "interaction contexts" in UsageCD, "interaction spaces" in Wisdom and "boundaries" in RUP (more recently "screens" in the new profile for web applications). For a comprehensive review of those concepts, please refer to [7].

However, all of the aforementioned techniques still leave a considerable gap between the inception level models of user intentions (task cases, use cases, scenarios and other requirements level models) and the concrete user interface. The center ellipse in Figure 1 illustrates this gap. A growing awareness of this conceptual gap lead Constantine and colleagues to develop a new language for User Interface (UI) specification, called Canonical Abstract Prototypes [1]. This language fills the gap between existing inception level techniques, such as the illustrated UML-based interaction spaces or visual content inventories, and construction level techniques such as concrete prototypes, as Figure 1 shows.

Informal tools supporting natural input (such as sketching) also allow faster modeling while communicating unfinished designs to clients and developers. This encourages exploration and promotes a systematic approach to building a UI. Two examples of this class of tools are SILK [4] and DENIM [6].

Our approach differs from others like [4, 6, 8] in the sense that we began with a simple, easy to use drawing application and then added the necessary formalisms, applying real-world notations (UML and Canonical Abstract Prototypes) while also seamlessly supporting multiple levels of abstraction (in terms of UI detail). The following section describes the languages and the semantic model of our tool. Section 3 briefly explains some of the tool's features and Section 4 draws some conclusions and future lines of research.

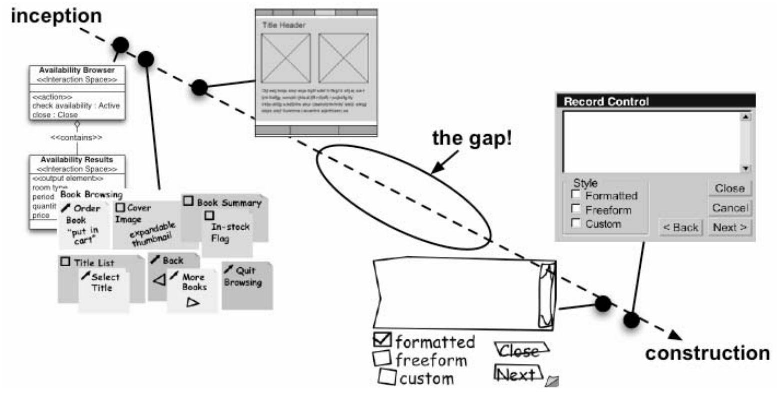


Fig. 1. Prototyping techniques from inception to construction (adapted from [1])

2 CanonSketch: The Languages

CanonSketch's main language is an extension to the UML for the design of interactive systems: the Wisdom profile [7]. The Wisdom UML view is at the highest level of abstraction: it is useful for drawing navigation and containment relationships without concern for details of layout or spatial positioning.

The HTML concrete view is at the lowest level of abstraction: it shows a partially-functional HTML prototype rendered in an embedded browser, thus allowing fast testing of the UI. The intermediate view is the Canonical Abstract Prototype view.

The symbolic notation underlying Canonical Abstract Prototypes is built from two generic, extensible universal symbols or glyphs: a generic material or container, represented by a square box and a generic tool or action, represented by an arrow. Materials represent content, information, data or other UI objects manipulated or presented to the user during the course of a task. Tools represent operators, mechanisms or controls that can be used to manipulate or transform materials [1]. By combining these two classes of components, one can generate a third class of generic components, called a hybrid or active material, which represents any component with characteristics of both composing elements, such as a text entry box (a UI element presenting information that can also be edited or entered).

Figure 2 shows a screenshot of a document being edited in CanonSketch. The left part shows the UML view and the right view shows the corresponding, synchronized, Canonical Abstract Prototype. We can see, for instance, that a containment relationship between two «interaction space» classes corresponds to a Canonical material nested inside a Canonical container. The complete, precise mapping between the three views can be found in [2].

3 CanonSketch: The Tool

All the existing approaches to model the presentation aspects of the user-interface rely on standard modeling tools to create and manipulate modified class models. However, modeling tools are well known to be difficult to use. This lack of usability of modeling tools is in fact responsible for weak adoption and a decline of gross sales of modeling tools (much less than expected and predicted from reputable sources such as the International Data Corporation). From an initial user-centered approach to create a new prototyping tool, we incorporated the semantics and formalisms necessary to provide automated generation of concrete prototypes.

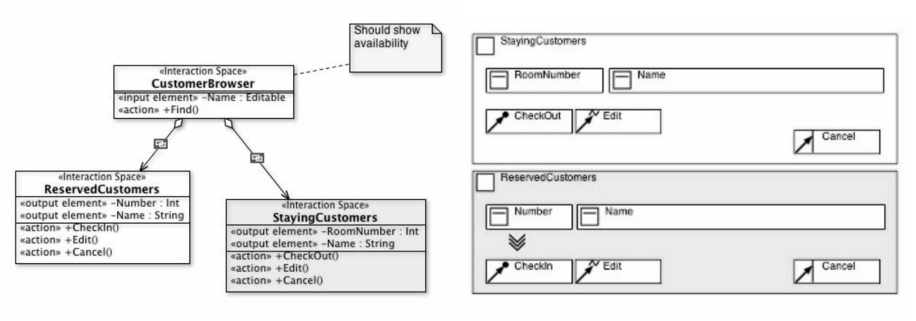

Should show availability
«Interaction Space» CustomerBrowser
«input element» -Name : Editable
«action» +Find()
«Interaction Space» ReservedCustomers
«output element» -Number : Int
«output element» -Name : String
«action» +CheckIn()
«action» +Edit()
«action» +Cancel()
«Interaction Space» StayingCustomers
«output element» -RoomNumber : Int
«output element» -Name : String
«action» +CheckOut()
«action» +Edit()
«action» +Cancel()
StayingCustomers
RoomNumber
Name
CheckOut
Edit
Cancel
ReservedCustomers
Number
Name
CheckIn
Edit
Cancel

Fig. 2. CanonSketch's screenshots, showing the same model at different abstraction views: the UML view (left) and the Canonical Abstract Prototype view (right)

What makes CanonSketch a developer-centered tool, besides the languages and the multiple, synchronized views, are its features, which include: color coding, a search box that quickly navigates the model, a grid layout option for helping the design of the spatial layout issues, tool palettes, a customizable toolbar, semantic checking and XMI export that allows tool interoperability at semantic level. The search box highlights all elements containing part of the string being typed, thus acting like a filter. Combined with color coding, this feature can be particularly useful for things such as quickly finding and classifying model elements (e.g. components coded as red are subject to change, classes coded blue represent interaction spaces, etc.).

Any change in the UML or Abstract Prototype view updates the other, e.g. if the user changes the name of an abstract component, that name is updated in the corresponding UML element. The HTML view is generated from the Abstract Prototype view. It is also possible to assign images or files to any kind of element.

Another aspect we recently added to the tool and has not been described yet is the possibility of using gestures (already implemented) and rough sketches (in development). The idea is to take advantage of natural input modalities in a flexible way that is critical for early development stages[1].

[1] For more information, please refer to [2] or visit the tools' website, which contains publica tions, videos and screenshots: http://dme2.uma.pt/canonsketch.

4 Conclusions and Future Work

This work shows that the UML semantic model can be used to support multiple levels of detail of an interactive system. The UML had a tremendous impact in software engineering but still remains quite far from achieving the promises of the late 90s. We still lack widely accepted notations to support user-centered development and user-interface design. We think that CanonSketch is a step towards that goal in bridging HCI and Software Engineering. We are attempting to change the way modeling tools are built and envisioned by focusing on achieving a flexible and usable tool instead of following formalism-centered approaches.

Our preliminary evaluations with using the tool have been positive, and the approach seems promising. However, the tool will have to be carefully evaluated and we plan to perform an extensive usability study in order to refine the tool and notations according to the study, add support for requirements modeling and achieve integration with application development. Another effort is being devoted to support real-time collaboration between several developers working on the same model. It has been recognized that software design is a highly collaborative activity but few tools support that cooperation. This support will be done in our tool by using Rendezvous [9] communication technology that will allow concurrent synchronous/asynchronous editing of models in a (hopefully) engaging and user-centered way.

References

1. Constantine, L. and Lockwood, L. A. D.: Software for use: a practical guide to the models and methods of usage-centered design, Addison Wesley, Reading, Mass, 1999.
2. Campos, P., Nunes, N. J., CanonSketch: a User-Centered Tool for Canonical Abstract Prototyping. In *Proceedings of the EHCI/DSV-IS'2004, International Conference on Engineering Human-Computer Interaction / International Workshop on Design, Specification and Verification of Interactive Systems*, Hamburg, Germany, 2004.
3. IBM, EMF-based UML 2.0 Metamodel Implementation: http://www.eclipse.org/uml2/.
4. Landay, J. and Myers, B.: Sketching Interfaces: Toward More Human Interface Design. IEEE Computer, pages 56-64, March 2001.
5. Myers, B., Hudson, S. and Pausch, R.: Past, Present and Future of User Interface Software Tools. ACM Transactions on Computer Human Interaction, 7(1): 3-28, March 2000.
6. Newman, M., Lin, J., Hong, J. I. and Landay, J. A.: Denim: an Informal Web Site Design Tool Inspired by Observations of Practice. Human-Computer Interaction, 18(3): 259-324, 2003.
7. Nunes, N. J., Cunha, J. F.: WISDOM: Whitewater Interactive System Development with Object Models, in Mark van Harmelen (Editor.), Object-oriented User Interface Design, Addison-Wesley, Object Technology Series, 2001.
8. Paulo Pinheiro da Silva and Norman W. Paton: A UML-Based Design Environment for Interactive Applications. In *Proceedings of the 2nd International Workshop on User Interfaces to Data Intensive Systems (UIDIS'01)*, E. Kapetanios and H. Hinterberger (Eds.), Zurich, Switzerland, pages 60-71, IEEE Computer Society, 2001.
9. Rendezvous Zero-configuration networking. Apple Computer: http://www.apple.com.

The AGEDIS Tools for Model Based Testing

Alan Hartman and Kenneth Nagin

IBM Haifa Research Laboratory,
Haifa University, Mt. Carmel 31905,
Haifa, Israel
{hartman, nagin}@il.ibm.com

Abstract. We describe the tools and interfaces created by the AGEDIS project, a European Commission sponsored project for the creation of a methodology and tools for automated model driven test generation and execution for distributed systems. The project includes an integrated environment for modeling, test generation, test execution, and other test related activities. The tools support a UML based testing methodology that features a large degree of automation and also includes a feedback loop integrating coverage and defect analysis tools with the test generator and execution framework.

1 Introduction

Model based testing is still not a widely accepted industry practice despite the existence of academic and industrial case studies (see e.g. [4, 5, 8, 11]) which discuss its advantages over traditional hand crafted testing practices. There are several reasons for this. Robinson [13] mentions the need for cultural change in the testing community, the lack of adequate metrics for automated testing, and the lack of appropriate tools and training material. The AGEDIS project is an attempt to remedy both the first and last of these obstacles to the wider adoption of model based testing. The AGEDIS project has created a set of integrated tools for the behavioral modeling of distributed applications, test generation, test execution, and test analysis. Moreover the AGEDIS tools are accompanied by a set of instructional materials and samples that provide an easy introduction to the methodology and tools used in model based testing. The case studies [5] undertaken by the AGEDIS partners show that not all of the tools are sufficiently mature for widespread adoption, but that they have all the necessary elements in place, that they are well integrated with each other, and that they provide a coherent architecture for UML based testing with well defined interfaces. The importance of this architecture lies in that it may be used as a plug and play framework for more or less sophisticated tools to be used as appropriate, and when more mature tools become available. As an example, the Microsoft tools for model-based testing come in two flavors, a light weight tool using visual modeling and straightforward test generation algorithms [12], and a heavy weight tool using a text based modeling language and sophisticated test generation based on model checking [10]. Either of these tools could be plugged in to the AGEDIS testing framework and take advantage of the features and facilities provided by the complementary tools. Similarly, other modeling languages may be substituted for the AGEDIS modeling language, simply by providing a compiler to the AGEDIS intermediate format for model execution. The importance of the AGEDIS tools and architecture lies not so much in the quality of one or other of the tools, but in the framework for integration of tools from different suppliers with different requirements and strengths.

2 Architecture

Figure 1 illustrates the software components of the AGEDIS framework (represented in the figure as rectangles), the user input artifacts (ovals), and the public interfaces (diamonds) for the use of tool makers.

The user inputs three pieces of information describing the system under test (SUT): a) the behavioral model of the system, b) the test execution directives which describe the testing architecture of the SUT, and c) the test generation directives which describe the strategies to be employed in testing the SUT. Both a) and b) are entered using a UML modeling tool equipped with the AGEDIS UML profile, whereas c) is input via an XML editor (e.g. XML Spy).

The behavioral model of the system under test is specified by the user in a combination of UML class diagrams, state diagrams, and object diagrams. The syntax and UML profile for this modeling language is described in [1]. The state diagrams are annotated with the IF action language defined in [2].

The test execution directives (TED) describe the testing interface to the SUT and give the mappings from the model's abstractions to the concrete SUT interfaces for control and observation. These are defined by an XML schema.

The test generation directives, describing the test strategy, are provided by the user either as test purposes using UML state diagrams, or as default test directives for global model coverage at varying levels of detail. These are also defined in [1].

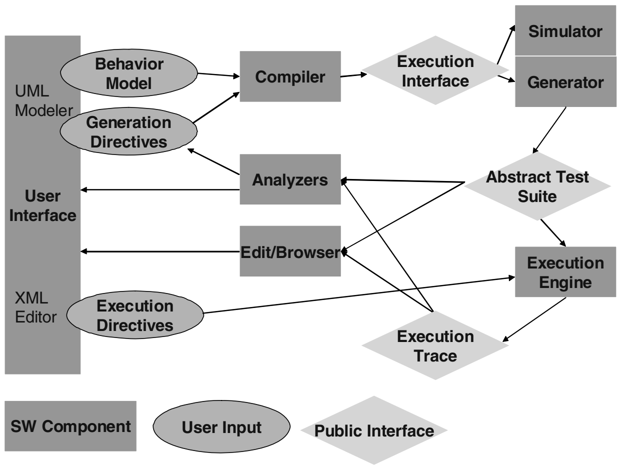


Fig. 1. The AGEDIS architecture

The three public interfaces for inter-tool communication are: a) the model execution interface, b) the abstract test suite, and c) the suite execution trace.

The model execution interface is defined in [2] and consists of the APIs used by both the test generator, and the model simulator. It incorporates the necessary data for simulation of the model of the SUT, including the controllable and observable features of the SUT.

Both the abstract test suite and the suite execution trace are defined by a single XML schema available at [3]. These two public interfaces provide all the necessary information to describe the test stimuli, and both the expected and observed responses by the SUT. The XML schema is a predefined abstract representation of all test suites and execution traces in a common format.

There are a number of components that have been integrated into the AGEDIS framework including: a) a UML modeling tool, b) a model compiler, c) a model simulator, d) a test generation engine, e) a test execution engine, f) a test suite editor and browser, g) a coverage analysis tool, h) a defect analysis tool, and i) a report generator. All of these tools are activated from a graphical user interface, which has management facilities for the various artifacts produced in the testing process.

The modeling tool (shown in Figure 1 as part of the user interface) can be any UML modeling tool with the ability to use the AGEDIS profile. The AGEDIS system uses a commercial UML Modeler with a convenient profile builder to produce an XML representation of the model.

The XML file is compiled, together with the test generation directives to create a combined representation of the model and testing directives in the IF 2.0 language. This representation is shown in Figure 1 as the execution interface.

The model simulator provides feedback on the behavior of the model in the form of message sequence charts describing execution scenarios. This simulator is an essential tool to enable the user to debug the model.

The test generator creates an abstract test suite consisting of test cases which cover the desired test generation directives. The test generator is based on the TGV engine [9], but with additional coverage and observation capabilities derived from the GOTCHA test generator [6].

The execution engine presents each stimulus described in the abstract test suite to the SUT, and observes the responses, waits for callbacks, and traps any exceptions thrown. The responses are compared with those predicted by the model, and a verdict is reached. The execution engine writes a centralized log of the test trace in the format defined by the test suite XML schema [3]. The execution engine also has the ability to run multiple instances of test cases and create stress testing from the functional tests created by the test generator. See [7] for details.

Both the test suite and execution trace can be browsed and edited by the AGEDIS editing tool. The tool is useful for composing additional manual test cases to add to the automatically generated test suites.

The coverage analysis and feedback tool which is integrated with AGEDIS is the functional coverage tool FoCus [14]. This tool enables the user to define a functional coverage model in terms of the methods and attributes of the objects in the SUT. FoCus itself provides coverage analysis reports, and AGEDIS has fitted it with a feedback interface, which creates test purposes for the generation of more test cases in order to increase the functional coverage.

The defect analysis and feedback tool was created for the AGEDIS tool set. It reads the suite execution trace and analyses the test cases which ended in failure. This was deemed a valuable addition to a testing framework featuring a large degree of automation, since large numbers of test cases are run automatically, and the same defect may be encountered many times in a given test suite. The defect analysis tool clusters test cases according to the similarities between the defects observed and the steps in the test cases immediately prior to the observation of the defect. The user can either view the clustering report or generate a new test purpose which will direct the test generator towards producing additional test cases which will replicate the characteristic defect of a cluster of test cases.

The report generator creates management documents describing the test cases, defects, models, and other artifacts of the testing process.

References

1. AGEDIS Consortium, AGEDIS modeling language specification, http://www.agedis.de.
2. AGEDIS Consortium, Intermediate Language 2.0 with Test Directives Specification, http://www.agedis.de.
3. AGEDIS Consortium, Test Suite Specification, http://www.agedis.de.
4. Becker P., Model based testing helps Sun Microsystems remove software defects. Builder.com http://builder.com.com/5100-6315-1064538.html.
5. Craggs I., Sardis M., and Heuillard T., AGEDIS Case Studies: Model-based testing in industry. Proc. 1st European Conference on Model Driven Software Engineering, 106-117. imbus AG December 2003.
6. Farchi E., Hartman A., and Pinter S. S., Using a model-based test generator to test for standards conformance. IBM Systems Journal 41 (2002) 89-110.
7. Hartman A., Kirshin A., and Nagin K. A test execution environment running abstract tests for distributed software Proceedings of SEA 2002 448-453.
8. Hartmann, J., Imoberdorf, C., and Meisinger, M., UML-based integration testing. Proceedings of ACM Symposium on Software Testing and Analysis (2000), 60- 70.
9. Jeron, T., and Morel, P., Test Generation Derived from Model-checking, Proceedings of CAV99, Trento Italy (Springer-Verlag LNCS 1633 1999), 108-122.
10. Microsoft Research – ASML Test tool, http://research.microsoft.com/foundations/AsmL/.
11. Offutt J. and Abdurazik A., Generating Tests from UML Specifications, Second International Conference on the Unified Modeling Language (UML99), 1999.
12. Robinson H., Finite state model based testing on a shoestring. Proceedings of STAR West 1999.
13. Robinson H., Obstacles and opportunities for model-based testing in an industrial software environment. *Proc. 1*st European Conference on Model Driven Software Engineering, 118-127. imbus AG December 2003.
14. Ur S. FoCus Functional Coverage Tool http://www.alphaworks.ibm.com/tech/focus.

Tools Exhibits

Alberto Rodrigues da Silva

INESC-ID and Instituto Superior Técnico,
Rua Alves Redol, nº 9 – 1000-029 Lisboa, Portugal
alberto.silva@acm.org

Abstract. Live demonstrations of cutting-edge systems were an important and exciting part of the UML2004 conference. The tool exhibits session provided an excellent opportunity where participants analysed and viewed relevant UML and MDA related tools in action and discussed these systems with their creators or distributors. The tool exhibits session took place during the main conference, from October 13 to 15, and included the following live demos: (1) "seCAKE: A complete CASE tool with reuse support", by dTinf; (2) "Making UML diagrams accessible for visually impaired programmers", by FNB; (3) "TAU Generation2", by Telelogic; (4) "IBM Rational Rose XDE Products", by Sinfic; and (5) "BridgePoint Development Suite", by Mentor Graphics. The tool exhibit contributions are described in this paper in the form of an extended summary. We briefly describe the related products according the data provided by their respective creators or distributors.

1 Introduction

In this paper we briefly describe the tools exhibits presented, from October 13 to 15, at the UML2004 conference in Lisbon. The tool exhibits session provided an excellent opportunity where participants analysed and viewed relevant UML and MDA related tools in action and discussed these systems with their creators or distributors.

The tool exhibits session included the following live demos:

- seCAKE: A complete CASE tool with reuse support, by dTinf;
- Making UML diagrams accessible for visually impaired programmers, by FNB;
- *TAU Generation2*, by Telelogic;
- IBM Rational Rose XDE Products, by Sinfic; and
- BridgePoint Development Suite, by Mentor Graphics.

The tools exhibits contributions are described in the following sections in the form of an extended summary. We describe the related tools according the data provided by their respective creators or distributors, or in same cases we also get information from the respective web sites. In the beginning of each section we provide contact information of related persons and organizations.

2 seCAKE: A Complete CASE Tool with Reuse Support (dTinf)

José M. Fuentes, Vicente García
dTinf, S.L. (Desarrollos para las Tecnologías de la Información), Madrid, Spain
{jmfuentes, vicente.garcia}@dtinf.es
http://www.dtinf.com

CAKE (Computer Aided Knowledge Environment) is a framework of tools, applications, and methodologies for identifying, classifying, retrieving, organizing, managing and reusing knowledge. The CAKE framework has been developed to manage, organize and reuse all different kinds of "knowledge assets" generated within an organization. The CAKE knowledge management and engineering environment, covering vertical applications, is based on a modern knowledge representation and classification schema called RSHP.

seCAKE is a computer-aided software engineering tool (CASE) designed for allowing software engineers to develop Information Systems using the Knowledge Management paradigm. Therefore seCAKE includes a whole set of enhancements for managing software as knowledge.

Tool Objectives

seCAKE has the following objectives: (1) A full coverage of the Software Development Process: unlike other CASE tools that are centered in UML, seCAKE covers other development stages such us requirements, estimation, testing… Furthermore, a trace system among all the project elements has been implemented. (2) Reuse support: by automatically index and retrieve software models using the RSHP [1] repository, seCAKE aims to open a new paradigm in software reuse. Together with the classical domain analysis and domain engineering techniques (also covered in CAKE) the indexing and retrieval capabilities of seCAKE enhance other classical classification schemas such as facets [2]. (3) A deep coverage of the UML standard: this objective is extremely important for retrieval and reuse purposes.

Main Features

The main features of the tool are the following:

- **High UML semantic accuracy:** in order to ease the retrieval of software models as much as possible, seCAKE tries to be complied with the UML standard. So, unlike many other tools, the core of the tool has been designed with the UML 1.5 metamodel in mind. seCAKE also includes a semantic checker that shows all the deviation of the current model with regards to the UML 1.5 metamodel and the UML wellformedness rules.
- **Advanced reuse system:** aside of the classical domain engineering techniques that are implemented in the CAKE tools, seCAKE includes a novel classification and retrieval module. This allows storing software models into the repository and querying it. Those queries are not textual ones, but model sketches (pieces of UML diagrams, one or more requirements…). During the query, the system will look into the reposi-

tory for those models that are close to the query. All those techniques allow an organization to enhance from an *ad-hoc* reuse maturity to a systematic one.

- **Requirements support:** including version management, trace with the rest of elements of the model (risks, function point items, UML model elements, test suites and anomalies). Also, thanks to the NLP (Natural Language Processing) capabilities of the tool, seCAKE allows to incorporate requirements automatically from a textual document.
- **Risk management:** aside of the trace, includes the management of the activities linked to risks, a priority model that shows the order of retirement and post-mortem information that gives an added value to the retrieval capabilities of the tool.
- **Project estimation techniques:** including the function point technique, the postarchitectural model of COCOMO II, and the POP (*Predictive Object Point,* developed by PriceSystems, http://www.pricesystems.com) estimation method.
- **UML models comparer:** that allows to easily compare the similarities and differences of two UML models.
- **Code and schemas generation:** including C# code generation, Java reverse and direct engineering and XML Schema generation.
- **Test cases management:** including test suits and test cases support, as well as anomalies support.
- **Report generation:** a project report could be generated with seCAKE. This report includes information about risks, requirements, estimation, test cases and especially UML.
- **Solution view:** that allows managing all the archives related to the project.
- **Modeling templates:** including the possibility of creating your own templates.
- **Knowledge sharing:** seCAKE will soon be deployed together with http://www.umlmodels.org. This web site will allow every seCAKE user to share their knowledge in two different senses: (1) seCAKE includes different forum threads; (2) the indexing and retrieval capabilities of seCAKE allow sharing software projects through the web site.

3 Making UML Diagrams Accessible for Visually Impaired Programmers (FNB)

David Crombie, Sijo Dijkstra, George Ioannidis
FNB, Amsterdam, Netherlands
projects@fnb.nl, george.ioannidis@tzi.de
http://projects.fnb.nl/, http://www.tzi.de

Computer programming is one increasingly important area where blind and visually-impaired people have been able widely to participate in the employment market and in the educational sector [3]. Computer code has traditionally been text-based, and therefore accessible to visually impaired people using assistive technology (such as screen readers). In recent years the growth of software engineering has led to an increase in the use of programming tools that use visually rich presentation methods to facilitate development by sighted programmers. One of these tools is the Unified Modelling Language, a language used for modeling across many fields. It reflects the dominant object-oriented programming paradigm, and is increasingly popular in education and the workforce.

UML diagrams consist of nodes and connections between them. A great deal of information is contained in UML diagrams, much of it in graphical formats, such as the style of arrow-heads, and this remains inaccessible in tactile format. Furthermore, the text parts of the UML diagrams need to be converted to Braille to be understood as a screen reader cannot read text contained within diagrams. Often when a tactile version of larger diagrams has to be produced, the information needs to be redrawn over many tactile drawings and thereby the connections and coherence are easily lost. This imposes practical limits on UML representation for blind people [4]. In addition, UML is designed to be a co-operative modeling tool and a dynamic source of information for a development team to annotate, amend and change as the team designs a system. To take advantage of these features, interaction with the content is clearly necessary and without access to these diagrams, visually impaired people are excluded. The EU funded TeDUB project (Technical Diagram Understanding for the Blind) has been working on access to technical diagrams via interfaces to structured information [5,6] and has now developed a successful tool to access UML.

Technical Description and System Architecture

The TeDUB system consists of two main parts, *DiagramInterpreter* and *DiagramNavigator*. DiagramInterpreter (semi-) automatically analyses existing diagrams from a number of formally defined domains and converts them into a representation that can be used by DiagramNavigator. DiagramInterpreter's core is the *knowledge processing unit*. It operates on a network of hypotheses and processes them incrementally until a semantic description of the whole diagram is found. The *image processing unit* analyses bitmap images and generates a first set of hypotheses based on the geometric information therein. Vector graphics files, which already contain explicit information about geometric primitives, can be used via DiagramInterpreter's SVG (Scalable Vector Graphics) import functionality. The *Annotator* allows a sighted user to interact with the interpretation process by inserting hypotheses manually and thus improving the quality of the interpretation as well as adding useful information not contained in the original diagram. All domain dependent aspects of DiagramInterpreter are externalised as formalised knowledge. Therefore, the system is designed to minimise the effort to incorporate a new type of diagram.

DiagramNavigator is the user interface component of the system and provides blind users with an interface to navigate and annotate these diagrams. It presents the diagram content obtained by DiagramInterpreter to the user. It also performs XSL transformation of XMI-format UML diagrams exported from UML design tools like Rational Rose or ArgoUML into the same TeDUB form, presented by the same user interface. The great advantage of this latter approach is that the information contained in the diagram is converted perfectly into the TeDUB format: the variable result of image analysis of bitmaps is avoided.

The information of the drawing is modelled as a hierarchical tree structure of nodes with information attached to each node. A node might be a Class in a Class diagram or an Actor in a Use Case diagram. The nodes can be navigated either hierarchically or as a collection of connected graphs (depending on the type of diagram), using standard application components so as to be screen-reader independent. Input is via the keyboard or an optional tactile tablet. This is combined with a number of alternative interfaces, such as 2D and 3D surround sound (optional) and a game force feedback joystick. The joystick is used as a simple input and output tactile device and allows representation of the spatial information in the diagram, such as the position of nodes or the connections between them. Access to the information is also allowed by means of a representation of the content as a text-only internally-hyperlinked document, which can be navigated with the cursor keys as a standard text document. Another approach to give access to the information is a set of functions based on task analysis of the diagrams' types and use that structure and represent information according to user needs. For example, a Use Case diagram allows the presentation of all the Use Cases separate from the diagram, useful for developers checking the required Use Cases against implemented functions. To reflect the co-operative and modeling needs of UML users, simple editing functions are available, including annotating, renaming and limited editing of node information. All these components are designed to form a cohesive whole, usable when the additional hardware is not available but consistent when it is. The devices used are widely-available, low-cost devices rather than relatively expensive special purpose devices. The system is intended to be affordable and usable without special equipment by users with their own familiar screen reader.

Application and Demonstration

Following extensive user studies across four countries, most participants expressed a very positive response to the system and felt it would be of significant value in educational and vocational environments. An interesting example was given by one participant, a software engineer, who had been made redundant when her department switched to UML, as she was unable to visualize the diagrams. A system like TeDUB would have resolved that problem. This stresses the importance of a tool to provide access to these kinds of diagrams. Taking the suggestions from this user study into account, the current version of the TeDUB system has significant potential to become a truly effective UML tool for visually impaired people.

4 TAU Generation2 (Telelogic Iberica)

Niklas Lagerblad
Telelogic Iberica, Madrid, Spain
niklas.lagerblad@telelogic.com
http://www.telelogic.com

TAU Generation2 (TAU G2) [7] is a family of model-centric and role-based tools that are among the first to implement the recently adopted UML 2.0 standard. The tool family consists of TAU/Developer for Software Engineers, TAU/Architect for Systems Engineers, and TAU/Tester for Test Engineers. TAU G2 builds on the model driven compilation technology perfected in TAU SDL Suite (a.k.a. TAU G1). TAU G1 proved that real-time software development can be automated using mature specifications languages such as Specification and Description Language (SDL) and Message Sequence Chart (MSC). Given that many of the advanced language features offered by SDL and MSC were adapted and incorporated into UML 2.0, there were compelling technical and market reasons to combine TAU G1's model driven compilation technology with UML 2.0 to produce TAU G2.

Main Features

TAU G2 provides the following features:

- **Precise and unambiguous system specification** – Engineers can visually specify systems using the precise, standardized and non-proprietary language of UML 2.0. This results in easy-to-understand, clear and unambiguous specifications.
- **Specification of behavior** – Whereas most system modeling tools allow only the specification of the system's architecture or structure, TAU G2 also allows engineers to visually specify the dynamic aspects of the system's behavior.
- **Automatic application generation** – TAU/Developer is the only tool that supports executable UML 2.0 models with behavioral specifications. Developers have access to pre-defined, verifiable code patterns that ensure high quality standards. With these capabilities, developers can automatically generate complete applications.
- **Dynamic model verification** – With fully controllable model simulation, engineers can verify their work in the analysis, design, and implementation phases. As a result, they can quickly locate and remove errors early when corrections are relatively easy and inexpensive.
- **Scalability** – Large scale systems can be specified and models can be mapped to how teams want to work, rather than having restrictions imposed by the tool. System architecture and behavior also can be modeled and viewed at the appropriate level of abstraction for the user.
- **Integrated requirements management via Telelogic DOORS®** – TAU G2 is integrated with Telelogic DOORS, the market leading requirements management solution.
- **Automated documentation via Telelogic DocExpress®** – TAU G2 is integrated with DocExpress, which provides automatic extraction and formatting of system or software application documentation.
- **Change and configuration management via Telelogic SYNERGY™** – SYNERGY provides change and configuration management for TAU G2 and related products.

Discussion

It's inevitable that the software industry will eventually mature, and catch up with other industries based on engineering and automation, such as the computer hardware industry. At some point during this maturation process, it will become common practice for software engineers to specify their products using an architectural blueprint language, such as UML 2.0.

During this evolution it will also become common sense for engineers to apply a model driven development approach, such as MDA. This approach will need to be supported by power tools, such as TAU G2, that faithfully and efficiently implement the blueprint language, so that it can automate the mapping transformations across the models that represent the various process phases.

What should we expect from Model Driven Architectures during the next decade? We should expect them to evolve from conceptual architectures into technical architectures that solve complex business and technology problems.

What should we expect from MDA tools, such as TAU G2? In general, we should expect progressively tighter integration with traditional Integrated Development Environments (IDEs), and improved integration with requirements management and testing tools. In the case of TAU G2, this means seamless integration with DOORS and TAU/Tester. DOORS requirements can already be visualized as UML elements and TAU/Tester test scripts are being updated to align it with the recently adopted UML 2.0 Profile for Testing.

This future model driven IDEs will allow developers to efficiently shift and downshift through all the abstraction gears associated with a full application lifecycle. In these high productivity development environments, programming code will likely devolve into a machine readable artifact that is rarely viewed by humans. Released from the drudgery of producing and maintaining low-level implementation code, software developers will be able to pursue more creative activities that return greater business value, such as architecture, analysis and design.

5 IBM Rational Rose XDE Products (Sinfic)

Paulo Figueiredo
Sinfic, Lisbon, Portugal
pfigueiredo@sinfic.pt
http://www.sinfic.pt, http://www.ibm.com

The IBM Rational Rose XDE product family [8, 9] combines the rich heritage of the award-winning IBM Rational Rose family with IBM Rational XDE, which extends your IDE with the world's most advanced software modeling capabilities. Though packaged and purchased together, Rose and XDE are installed separately. They can be used in combination, with some limitations, but most users will benefit from primarily using one or the other.

IBM Rational Rose XDE Developer editions offer software designers and developers a rich set of model-driven development and runtime analysis capabilities for building quality software applications. They offer complete visual design and development environments that address the needs of organizations targeting both J2EE-based and NET-based systems. Our solution allows users to work inside the included Eclipse IDE, or it can be installed into the IBM WebSphere Studio Application Developer and Integration Edition IDEs, and Microsoft Visual Studio .NET. Rational Rose is also included to integrate with Microsoft Visual Studio and other leading Java platform IDEs. The IBM Rational Rose XDE Developer products extend your development environment or integrate with the one you are already using.

Rational Rose XDE Modeler enables architects and designers to practice model-driven development with the Unified Modeling Language (UML). Such users can produce platform independent models of software architecture, business needs, reusable assets, and management-level communication. Industry standard UML support and a powerful pattern engine allow users to create a semantically rich application architecture that meets business needs and is readily understood by the development team. Architects and designers can use Rational Rose XDE Modeler's multi-model support to separate concerns of analysis, architecture, design and implementation. Developers can use architectural models and patterns as the basis for implementation, thereby accelerating the development of applications to conform to their architecture. Further, features such as free-form modeling, Web publishing and reporting allow users to share architecture and designs with all stakeholders, whether or not the stakeholders use Rational Rose XDE Modeler.

Rational Rose XDE Developer also allows data architects and DBAs to create logical and physical data models for DB2, Oracle, Sybase, and SQL Server databases. Architects can follow a top-down approach, creating a logical model of data requirements, transforming it into a physical database design, and then deploy that design to a database. Or, start from an existing database and reverse engineer the schema into a physical data model. Sophisticated "Compare and Synch" capabilities allow you to compare a physical data model to a database, and reconcile differences. Further, since the data modeling capabilities share the same environment as the application modeling environment, it is easy to keep data models and application models in synch.

No Need to Switch Between Tools for Design/Development

IBM Rational Rose XDE products extend your development environment or integrate with the one you are already using. The developer doesn't need to switch between the IDE and the visual modeling tool. Purify plus it's also integrated into websphere studio an Microsoft Visual Studio .Net, that allows use the capabilities of this performance and runtime analysis tool without leaving the IDE.

Main Features

The relevant features of the IBM Rational Rose XDE products are the following ones:
- Model-driven development with UML support
- Roundtrip engineering Java,C++, and Visual Studio languages
- Automatic or on demand model-code synchronization
- User-definable patterns and code templates
- Runtime analysis including visual execution trace
- Assisted modeling
- Multiple model support for Model-Driven Architecture
- Free-form diagramming
- Logical and physical database design
- Web publishing and reporting

6 BridgePoint Development Suite (Mentor Graphics)

Thomas Ulber
Mentor Graphics, Munchen, German
thomas_ulber@mentor.com
http://www.mentor.com

The Nucleus BridgePoint Development Suite [10] accelerates the development of real-time, embedded, technical, and simulation systems. Nucleus BridgePoint provides the most complete and productive environment for Agile MDA (Model Driven Architecture) [11] and the development of Executable and Translatable UML models. It has been used to develop hundreds of the most demanding systems including flight-critical launch vehicles, life-critical medical systems, large fault-tolerant distributed telecom systems, highly resource-constrained consumer electronics, and large-scale distributed discrete-event HLA simulation systems.

Project Technology founders pioneered the concepts of Executable and Translatable UML and have shaped the OMG standards that make UML executable. Accelerated Technology continues Project Technology's leadership role by delivering the benefits of MDA automation to development teams today.

Agile MDA – Executable and Translatable UML

Agile MDA provides a unique opportunity to accelerate the development and improve the quality of real-time, technical, and simulation systems. OMG-compliant Executable and Translatable UML (xtUML) provides the basis for Agile MDA and its significant benefits.

xtUML Platform Independent Models (PIMs) completely and concisely describe what the system does and are fully testable and executable. The three orthogonal system aspects are defined with Class Diagrams (Data), State Charts (Control) and OMG-compliant Object Action Language (Processing). Domain Package Diagrams provide support for effective subject-matter partitioning and system scale-up. Through early PIM testing and defect elimination, system quality is dramatically increased, and downstream integration, test and maintenance activities are streamlined.

xtUML PIMs are automatically translated, by customizable model compilers comprised of translation rules and patterns, to generate 100% complete target code. The generated code directly reflects both the application behavior defined and tested in the PIM, and the design and implementation specifics defined and tested in the model compiler. Changes to the application defined by the PIM or to the software architecture defined in the model compiler are automatically reflected in the system's generated code.

Benefits of effective automation and implementation of the Agile MDA process include: accelerated development and maintenance, greatly increased system quality, effective performance and resource optimization, streamlined platform migration, and large-scale reuse.

Nucleus BridgePoint Development Suite

The Nucleus BridgePoint Development Suite provides complete support for Agile MDA and the construction, debug, test, management, and translation of Executable and Translatable UML (xtUML) PIMs. The Nucleus BridgePoint Development Suite provides:

- Guided development of high-quality xtUML PIMs.
- Early (pre-code) execution, debug and test of xtUML PIMs.
- Customizable translation of xtUML PIMs into target-optimized 100% complete code.

- Model-level test and debug of complete or partial systems comprised of generated and non-generated code.
- Powerful performance-tuning and system-resource optimization.
- Effective reuse of xtUML PIMs and PIM components across multiple releases, products and product lines.
- Robust model configuration management including concurrent branches, overlapping configurations, and versioned domains, subsystems, and class statecharts.
- Multi-user, heterogeneous network support.
- Multi-level subject matter partitioning for effective project scale-up and accelerated iterative development.
- Nucleus BridgePoint Model Builder.
- Building quality into UML models that execute and translate.

7 Conclusions

The tools exhibits session, integrated in the main activities of the UML2004 conference, was a key opportunity to gather both the tools creators or distributors and interested people from the academic and industry world. It was an exceptional occasion to see live demonstrations of cutting-edge systems.

In an overview analyse we verify that the number of features and level of complexity of the most part of the shown tools is very high.

For example, the tools suite from IBM (e.g., Rational Rose XDE [8], Rational Software Architect [9]), from Telelogic (e.g., TAU G2, Doors, Synergy) or even from dTinf (e.g., seCAKE) support several activities of the software development process. Predominantly, they better support the following activities: requirement engineering, visual modelling and development. A common aspect shared by these tools is their support for visual modelling in UML and also an emergent set of features concerning the MDA concept.

The sophistication level of the current state presented in these tools is something remarkable, which raises several questions, such as: What is the necessary effort to learn how to use them productively? Is it possible and how can we tailor or customize these tools according business needs or specific projects needs (e.g., team size, project complexity)? Is it possible to integrate these tools with others (such as IDEs or project management tools)? How effort does this require?

In a different way, the BridgePoint Development Suite addresses the precise domain area of real-time, embedded, technical, and simulation systems. BridePoint applies innovative and interesting ideas around the Agile MDA and xtUML (executable and translated UML) model concepts. Based on these concepts, BridgePoint provides a set of tools that supports the design, debug, test, animation, and translation of xtUML platform independent models.

Finally, the TeDUB system is the result of the EU funded TeDUB ("Technical Diagram Understanding for the Blind") project and addresses the ability to make UML models accessible for visually impaired people (particularly, technical people). Definitely, TeDUB is a system for a very narrow but important group of people, and we believe that it is still in an initial stage of development.

References

1. Lloréns, J., Morato, J., Génova, G.: RSHP: an information representation model based on relationships. Springer Verlag in the LNCS series; Soft-Computing in Software.
2. Prieto-Díaz, R., Freeman, P.: “Classifying software for reusability”. IEEE software (January 1987) 6-16.
3. C. Baillie, O. K. Burmeister & J. H. Hamlyn-Harris, “Web-based Teaching: Communicating Technical Diagrams with the Vision Impaired” (available at http://opax.swin.edu.au/~303207/OZeWAI20031.html).
4. M. Horstmann, C. Hagen, A. King, S. Dijkstra, D. Crombie, D. G. Evans, G. T. Ioannidis, P. Blenkhorn, O. Herzog, Ch. Schlieder (2004) 'TeDUB: Automatic Interpretation and Presentation of Technical Diagrams for Blind People', Proceedings Conference and Workshop on Assistive Technologies for Vision and Hearing Impairment, University of Glasgow.
5. P. Blenkhorn, D. Crombie, S. Dijkstra, G. Evans, B. Gallager, C. Hagen, M. Horstmann, G. Ioannidis, A. King, M. Magennis, H. Petrie, A. O’Neil, C. Schlieder & J. Wood, Access to Technical Diagrams for Blind People, AAATE, In Craddock, McCormack, Rielly & Knops (Eds.), Assistive Technology – Shaping the Future (Proc. AAATE 2003), pp 466 – 470, 2003.
6. H. Petrie et al, “TeDUB: A System for resenting and Exploring Technical Drawings for Blind People”, In K. Miesenberger, J. Klaus, & W. Zagler (Eds.), Computers Helping People with Special Needs, Proc. 8th ICCHP, Lecture Notes in Computer Science, No. 2398, Springer, pp 537 - 539, July 2002.
7. Telelogic *TAU Generation2*. http://www.telelogic.com/products/tau/tg2.cfm
8. IBM *Rational Software Architect*. http://www-306.ibm.com/software/awdtools /architect/swarchitect/index.html
9. IBM *Rational XDE Developer*. http://www-306.ibm.com/software/awdtools/developer/rosexde/
10. Mentor Graphics *BridgePoint Development Suite*. http://www.mentor.com/products/ embedded_software/nucleus_modeling/nucleus_bridgepoint/index.cfm
11. OMG. 2003. *MDA (Model Driven Architecture) Guide Version 1.0.1*. www.omg.org/mda.

Author Index