Alan Hartman David Kreische (Eds.)

Model Driven Architecture – Foundations and Applications

First European Conference, ECMDA-FA 2005
Nuremberg, Germany, November 7-10, 2005
Proceedings

Springer

Volume Editors

Alan Hartman
IBM Haifa Research Laboratory
Model Driven Engineering Technologies
Haifa University Campus
Mt. Carmel, 31905, Haifa, Israel
E-mail: hartman@il.ibm.com

David Kreische
imbus AG
Kleinseebacher Str. 9, 91096 Moehrendorf, Germany
E-mail: david.kreische@imbus.de

Library of Congress Control Number: 2005935882

CR Subject Classification (1998): C.2, D.2, D.3, F.3, C.3, H.4

ISSN 0302-9743
ISBN-10 3-540-30026-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-30026-7 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11581741 06/3142 5 4 3 2 1 0

Preface

The European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA) is a new conference dedicated to the study and industrial adoption of the model-driven approach to software engineering. It has grown out of a number of workshops and smaller conferences in the area of model driven development and model driven architecture (MDA^{TM}). The conference is dedicated to providing a forum for cross fertilization between the European software industry and the academic community. We aim to present the industrial experience and highlight the pain points of industry in order to promote focused academic research that will bring real value to society. At the same time, we hope to challenge industry leaders to conduct a realistic appraisal of the emerging technologies presented by academics, consultants, and tool vendors, and eventually to adopt the model driven approach.

The conference provides both a forum for the papers judged as being of the highest quality and a venue for workshops, tutorials and tool exhibitions on model driven software engineering. This year, we are host to five workshops and four tutorials in subject matter ranging from the highly theoretical to the practical industrial aspects of MDA and a tool exhibition featuring nine commercial and seven open source or academic tools. This volume contains nine papers from the applications track and fifteen from the foundations track, chosen from 82 submitted papers. These works provide the latest and most relevant information on model driven software engineering in the industrial and academic spheres.

I would like to express my thanks to all the members of the steering committee, the program committee, and the referees, who gave freely of their time and wisdom to make this conference a success. The ECMDA-FA is supported by the European Commission's Information Society Technologies (IST) initiative, and by the Object Management Group (OMG).

November 2005

Alan Hartman
Program Chair
ECMDA-FA'2005

Organization

Steering Committee

Program Chair:	Alan Hartman (IBM)
Local Arrangements Chair:	David Kreische (imbus AG)
Workshop Chair:	Arend Rensink (Twente U)
Tools and Tutorials Chair:	Jos Warmer (Ordina)
	Uwe Assman (Dresden TU)
	Asier Azaceta (ESI)
	David Akehurst (Kent U)

Programm Committee

Jan Aagedal
Mehmet Aksit
Sergio Bandinelli
Mariano Belaunde
Jean Bezivin
Xavier Blanc
Manfred Broy
Krzysztof Czarnecki
Miguel A. de Miguel
Philippe Desfray
Reiko Heckel
James J. Hunt
Jean-Marc Jezequel
Anneke Kleppe
Richard Paige
Bernhard Rumpe
Bran Selic
Marten van Sinderen
Gerd Wagner
James Willans
Jim Woodcock

Additional Reviewers

Andreas Bauer
Machiel van der Bijl
Peter Braun
Phil Brooke
Maria Victoria Cengarle
Anthony Elder
Eitan Farchi
Boris Gajanovic
Adrian Giurca
Roy Grønmo
Hans Grönniger
Wilke Havinga
Jan Jürjens
Dave Kelsey
Mila Keren
Andrei Kirshin
Holger Krahn
Sergey Lukichev
Keith Mantell
Frank Marschall
Tor Neple
Dimitrios Kolovos
Shiri Kremer-Davidson
Jon Oldevik
Sergey Olvovsky
Jonathan Ostroff
Fiona Polack
Gerhard Popp
Martin Rappl
Julia Rubin
Thomas Roßner
Martin Schindler
Yael Shaham-Gafni
Zoe Stephenson
Alexander Wißpeintner

Table of Contents

Applying MDA to Voice Applications: An Experience in Building an MDA Tool Chain

Maria José Presso and Mariano Belaunde

France Telecom, Div. R&D,
2, Av Pierre Marzin, 22307 Lannion, France
{mariajose.presso, mariano.belaunde}@francetelecom.com

Abstract. Before a development project based on MDA can start, an important effort has to be done in order to select and adapt existing MDA technology to the considered application domain. This article presents our experience in applying MDA technology to the voice application domain. It describes the iterative approach followed and discusses issues and needs raised by the experience in the area of building MDA tool chains.[1]

1 Introduction

Interactive voice-based applications are specific telephony applications that are designed to allow end-users to interact with a machine using speech and telephone keys in order to request a service. The interaction – called a dialog –typically consists of a state machine that executes the logic of the conversation and that is capable of invoking business code which stands independently of the user interface mechanism – could be web, batch or speech-based. Because state-machines can be specified and modelled formally, it is possible to design a tool chain that automates large amounts of the dialog implementation. The application of model-driven techniques to this domain is without any doubt very promising. However, the question that arises is about the methods and the cost of building a complete environment capable of taking full advantage of models: not only ensuring automated code production but also offering user-friendly interfaces to designers, model simulation and test generation.

A general methodology for MDA-based development has been defined in [1]. The authors define the main phases, and make a distinction between preparation and execution activities: execution activities refer to actual project execution, during which software artefacts and final products are produced, while preparation activities typically start before project execution, and setup the context that allows the reuse of knowledge during the project. The preparation activities can be seen as selecting and adapting existing generic MDA technology in order to define an MDA approach for the considered application domain and provide an appropriate tool chain.

[1] The work presented here has been partially carried out within the MODELWARE project. MODELWARE is a project co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2002-2006). Information included in this document reflects only the author's views. The European Community is not liable for any use that may be made of the information contained herein.

Whereas [1] gives a general description of preparation activities and their chaining, there is currently little or no guidance available for them. In the current state of MDA technologies, these preparation activities demand an important effort which is not paid of by the first project and should be shared among a set of projects within the same domain. In particular, preparation has an impact on the first application project that follows it, as this first project necessarily requires iterations in the preparation activities.

This work presents our experience in building an MDA approach for the voice application domain and finishes by a discussion on the encountered issues and expectations.

2 MDD Preparation for the Voice Application Domain

According to [1], the preparation activities are divided into the *preliminary preparation phase*, the *detailed preparation phase*, and the *infrastructure set-up phase*. The *preliminary preparation* comprises the identification of the platform, the modelling language identification, the transformation identification and the traceability strategy definition. *Detailed preparation* comprises specification of modelling languages and specification of transformations. *Infrastructure setup* includes tool selection and metadata management.

In our project, these activities where performed in an iterative and incremental way, in order to better suit the needs of the users of the MDD environment involved in the execution phases, like voice dialog design, business application coding and functional testing. These users apply the tool facilities constructed by the preparation activities to produce the voice applications (the tool facilities are described later).

The preparation took place in three main stages. In each phase, some preparation activities were executed, together with some validation activities involving future users of the tool-chain, mainly service designers. The rest of this section presents the stages we followed.

2.1 Stage 1: Definition

In the first part of this stage, the current process of voice application creation was analysed and the integration of MDD techniques to this process was studied in order to identify the requirements for the voice development environment (VDE).

From this study, the following roles and their corresponding scenarios of use of the VDE were identified:

– the **service designer** uses the VDE model editor and simulator to model and simulate iteratively the dialogs of the application,
– the **usability practitioner** uses the VDE simulator to perform usability expertise on the dialogs of the service and he uses the VDE model editor to correct the dialog model,
– the **internal customer** (project owner) uses the VDE simulator a prototype of the service, to validate dialog design,
– the **service implementer** implements the service in the target platform, ideally by completing the skeletons generated by the modelling tool,
– the **service validator** produces the conformance test cases using the VDE test generation tool.

In order to serve as a conceptual basis for the VDE, a meta-model for platform independent modelling of voice applications was defined and UML 2 was chosen as a concrete syntax. Thus, a UML 2 profile for voice application models was defined[2].

Although the choice of UML for the concrete syntax may seem obvious, UML 2 being "the" modelling language standard, we will see later that this choice induces a significant cost in the infrastructure setup phase. For this reason, it is important to recall the rationale behind his choice:

- Voice application logic can easily be assimilated to a reactive state machine: the application reacts to user input such as voice and telephone keys, and produces output for the user: the vocal messages. The concepts of states and transitions are used in the voice application meta-model and supported by UML.
- Voice applications usually interact with the enterprise's information system. As UML is used as a modelling language in the information system domain, using the same language for voice applications allows to seamlessly integrate information system models with voice application models.
- Communication services are becoming integrated and multimodal. The use of a standard, largely used notation is expected to favour future integration with other services and modalities.
- Existing modelling tools and skills can be reused.

Also at this stage, the architecture of the VDE was defined (see Figure 1) and some of the tools involved were selected. A UML tool was chosen to play the role of model editor and model repository and the criteria for its selection were defined. Among the criteria defined, the ones that differentiated the tools were : i) the support for UML 2 transition oriented syntax for state machines (this notation had been found easier to read by users than the state oriented syntax), ii) the support for rigorous syntax checking (in particular for actions) and iii) model simulation/execution capabilities. TAU G2 from Telelogic was chosen as modelling tool.

Following this first round of preparation activities, we conducted some experiments in order to determine the ability of the profile to capture the intended service logic, verify that service designers could feel comfortable with the tool and to identify the necessary adaptations to the modelling tool (i.e. specific functionalities necessary to better support the voice application profile). These experiments consisted in modelling some existing services with the proposed profile and tool. Modelling was initiated by a modelling expert and a service designer working in pairs and finished by the service designer.

As a result of this phase, we could see that the proposed profile captured most of the necessary elements to describe a voice application, but it should be enhanced to support executable modelling of messages and the definition of grammars for voice recognition. Designers (who are not modelling experts) could get familiar with the modelling tool with a fair amount of effort. The main need for tool adaptation that came up was that a high level of support for the specification of messages allowing to reuse message parts, as well as facilities to read them during the dialog specification task. Also, high level commands for the creation of the other domain elements should

[2] This profile was later used as a basis for a submission to the OMG's RFP for a Metamodel and UML Profile for voice applications [3].

be available (such as creating a dialog). The restitution of the specification in the form of a document should be optimized in order to limit its volume and improve readability, hyperlink navigation should be available in the documents.

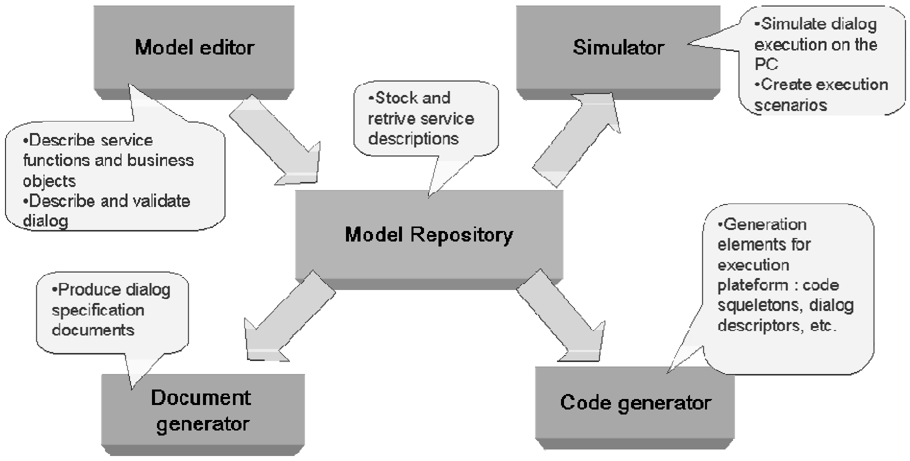

Model editor
Simulator
•Simulate dialog execution on the PC •Create execution scenarios
•Stock and retrive service descriptions
•Describe service functions and business objects •Describe and validate dialog
Model Repository
•Generation elements for execution plateform : code squeletons, dialog descriptors, etc.
•Produce dialog specification documents
Document generator
Code generator

Fig. 1. Architecture of the MDD Voice Development Environment (VDE)

2.2 Stage 2: VDE Development – Iteration 1

The second stage consisted mainly in the development of a first version of the tool chain, offering assistance for the creation of dialog modelling elements (dialogs, messages, recognition interpretation concepts, etc.), documentation generation and a limited form of dialog simulation. This version was developed using scripting and code generation capabilities provided by the modelling tool. Concretely, it appears as a plug-in to the modelling tool and a separate telephone-like GUI connected to a text to speech engine, that allows the designer to execute the dialog logic of the modelled service and evaluate its appropriateness, its ergonomics, etc.

After the development of this first version, a second row of experiments took place, which consisted in using the tool chain to enhance the service models produced in the first stage, and use the document generation and simulation functionalities for this models.

Although this first version of the tool chain was very promising and showed that useful functionality for service creation could be offered, it presented some limitations: the GUI for modelling assistance that could be developed through scripting was limited and not satisfying from the point of view of ergonomics; the service simulation was not able to propose the possible inputs in a given situation, neither to go arbitrarily back and forth into the execution tree. The scripting technique used for development posed maintainability issues and was not appropriate to support a growing software.

At the same time, studies where carried out about simulation and test generation for voice services. This studies showed that existing simulation technologies based on the IF language[2] provided the necessary level of support to build a simulator for voice services that overcomes the limitation of the method employed in the first version.

At the end of this stage the decision was taken to build a more industrial version of the tool chain based on a programming language (rather than scripting), to offer richer GUI capability in particular for message creation, and to provide the service simulation functionality through model simulation techniques, rather than code generation.

2.3 Stage 3: VDE Development – Iteration 2

This stage started by the definition of the architecture for the modelling tool plug-in, and the choice of the implementation technology. The main characteristic of this architecture was the definition of a layer that provides a view of the underlying UML model in the terms of the voice application metamodel. This layer implements an on-the-fly bi-directional transformation between UML and the voice application metamodel. This layer provides an adapted API and is used as a basis to develop the GUI and a set of generators that implemented various model transformations. Also, the architecture proposed a way for simple integration of the different generators in the plug-in. The generators provided at this stage were:

- document generators, which produce documents according to different templates and in html and MS Word formats. These generators use an intermediate XML generation phase, followed by XSLT transformations,
- an XMI generator, which exports the model in the terms of the voice application metamodel. This generator uses the adaptation layer API, and was automatically generated (and re-generated as needed) from the voice application metamodel,
- a generator that produces an IF model for service simulation and test generation,
- a code generator having as target an n-tier architecture using VoiceXML. The generated code executes in the application server tier and produces on-the-fly the presentation pages in VoiceXML. The generated code integrates in a framework (which was also developed in this phase), that provides the basis for the execution of a dialog state machine and VoiceXML generation.

The first three generators above were directly integrated into the plug-in, in order to facilitate the installation of the toolkit in the user's workstation and its use by the service designers, while the last on is external and uses the results of the XMI export.
Something important to note about this stage is that the metamodel was called to change often, as the implementation of transformations asked for corrections or improvements. As different transformations were developed in parallel, the changes asked by one of them had an impact on the others.

2.4 Stage 4: Pilots

The last stage is that of pilot projects. These are the first projects using the MDD chain (in the terms of [1], the first runs of the "execution" phase). The stage is still in progress at the time of writing. During this phase, iteration with preparation activities goes on, mainly to adapt the code generator to the project's target platform and to add extra functions asked by the pilot projects. An important effort in this stage is spent on user training and support.

3 Discussion

The result of the preparation activities described in the previous section is a process and a tool chain providing a high degree of automation. Starting from the PIM model, the tool chain produces automatically a simulation of the dialog, functional test cases, and the executable code for the dialog logic, and a is good representative of the MDA vision. However, these activities consumed an important effort, that should be shared by several projects. In this section we briefly discuss some of the issues and expectations that come from our experience in building this MDA tool chain.

In our experiment, we encountered a strong user's demand to have a rich GUI for modelling in terms of the domain vocabulary. This appeared as a critical issue for the adoption of the tool chain. As UML had been chosen as a concrete syntax, this request lead to important extensions to the modelling tool in the form of a plug-in. The ability to extend the modelling tool using a full-fledged programming language was necessary to develop the required GUI and to apply good engineering practices (such as the MVC pattern) to this development.

The above issue comes to the famous problem of whether a general-purpose modelling tool should be specialized or whether the tool should be built from scratch. Beyond tool usage is the question whether the specific language for the considered domain – in our case voice dialog definition – has to be built on top of an existing language – like UML, or a new language should be defined – typically using MOF or an XML schema. It is easy to adhere to the principle of maintaining the distinction between the abstract syntax (the domain metamodel) and the concrete syntax (given by a UML profile or a textual notation) since it provides potentially much more freedom – ability to use various concrete syntaxes - and is more comfortable for domain designers – since the vocabulary used is directly the one of the domain. However, as our experiment has demonstrated, maintaining this distinction potentially induces a high cost to the development of the tool chain. In our experiment, we used an "API adaptation" technique which allows to program a specific GUI and model transformations within the UML tool by using an API dedicated to the domain metamodel – instead of using the general-purpose UML-based API. This technique presents interesting advantages from the engineering point of view : i) the knowledge about the mapping between UML and the domain metamodel is localized, ii)the coding of the GUI is simplified, since the complexity of UML is hidden iii) the model transformations can be implemented in domain terms and are thus facilitated, and iv) the XMI generator exporting the model in the voice metamodel terms can be produced and updated automatically from the metamodel itself. However, one of the important problems encountered at this level was the instability of the metamodel, which in general changed more often than the graphical and the textual notation. Ideally, to solve the instability problem, the mapping between the metamodel and the concrete syntax should be defined in one single place and then the "API adaptation" generated automatically. This is indeed easy to say but not necessarily easy to put to work, especially when the evolving API is already used in various places.

To conclude with the "API adaptation" technique, our experiences showed us that this technique has good properties, but is costly when the metamodel is not stable and there is no specific advanced support for maintaining mapping coherence. At the moment we don't know what the cost would be without this technique, may be it would be higher. In other words there is a need for tools that will offer an explicit support of API adaptation. The current notion of UML profile is not sufficient as a specialization mechanism since it only addresses notation customization but not API customization.

Concerning metamodel stability, the implementation of code generation and simulation model generation asked for much more changes to the metamodel than documentation generation and GUI development. In our experiment this meant that the metamodel changed more during stage 3 than stage 2. A possible conclusion from this observation is that transformation development should happen earlier in the preparation phase, in order to rapidly stabilize the metamodel, before heavy development such as tool adaptation take place. In order to put this scheduling in place, we need a way to easily produce metamodel compliant models to serve as an input to test model transformations. Tools able to rapidly produce a friendly GUI or text notation from the metamodel definition could be very useful in this situation.

Another interesting issue is about reuse of metamodel patterns. The PIM Voice metamodel reuses various common constructs that are already found in UML, such as the differentiation between operation definition and operation call – which has a variant in dialog definition versus sub-dialog invocation concepts. The UML2 infrastructure metamodel is currently defined using an extensive package decomposition and the merge mechanism is intended to pick the packages that are needed. In theory, the semantics of the imported merged concepts is preserved. UML Profiles is a restricted way to perform metamodel extension with semantics preservation of the reference metamodel. Ideally, we would like to see a tool that will allow plain metamodel extension that will ensure – when applicable – semantic preservation of the reused patterns. This would be an option to the "API adaptation" technique described above that could avoid maintaining the distinction between the concrete and the abstract syntax.

A last issue concerns the need of providing an environment that integrates the different parts of the tool chain in a single workspace to improve easiness of use and installation. This is still a difficult matter with existing MDA technology. In our experiment this need led us to integrate some of the transformations directly into the modelling tool plug-in, where a model transformation engine would be more appropriate from an MDA point of view. In spite of this integration effort, we still needed to deploy additional tools and technology in the users' site and decided to host code generation in a web server, to simplify the installation on the developers' site. Better integration of MDA tools is necessary to easily build and deploy an integrated MDD environment for a given domain.

4 Conclusions

Preparation activities are an important and unavoidable phase in an "MDA development trajectory". We have presented an experience which applied an iterative approach to preparation activities and led to an MDA tool chain with a high level of automation. These preparation activities need an important effort and thus impact on MDA ROI. Better specific tool support and integration are necessary to lower the cost of this phase and improve the integration level of the resulting tool chain.

Acknowledgments

Several people contributed to the design and implementation of the presented tool chain : Miguel Alabau, Fabrice Dubois, Gregoire Dupé, Mikaël Marche, Sébastien Poivre (France Telecom Div R&D) , Charles-Henri Jurd (Telelogic), Eric Bousquet and Cedric Le Maitre (Capgemini).

Special thanks come to Gregoire Dupé and Sébastien Poivre for interesting discussion on issues and possible improvements on the development process of an MDD tool chain.

References

1. A.Gavras, M.Belaunde, L. Ferreira Pires, J. P.A Almeida. Towards an MDA-based development methodology for distributed application, in EWSA 2004 : 230-240.
2. M. Bozga, S. Graf, I. Ober, I. Ober and J. Sifakis. Tools and Applications II: The IF Toolset. In Flavio Corradinni and Marco Bernanrdo, editors, Proceedings of SFM'04 (Bertinoro, Italy), September, 2004 LNCS vol. 3185, Springer-Verlag
3. Joint Initial Submission to the UML Profile and metamodel for voice-based applications RFP. August 2004.

MDA, Meta-Modelling and Model Transformation: Introducing New Technology into the Defence Industry

Tony Bloomfield

SELEX Sensors and Airborne Systems Ltd. (formerly BAE Systems Avionics Ltd), Crewe Toll, Ferry Road, Edinburgh EH5 2XS, UK
tony.bloomfield@selex-sas.com

Abstract. The paper discusses some practical examples of how Model Driven Architecture (MDA) technology is being applied to some vital issues in the development of avionics systems. A study primarily aimed at addressing the issue of Software Method and Tool obsolescence was conducted by a number of BAE Systems' sites and York University. It investigated model transformation from legacy Teamwork[1] models to UML models. It then went on to investigate the re-modelling of a component of a legacy avionics system in executable UML, the development of an Ada code generator, and the integration of the resulting auto-generated code into the embedded system without any degradation of functionality and performance. Another Anglo-French study was conducted to investigate software development methods and tools for the challenging Integrated Modular Avionics architecture. Both of these studies contained themes regarding the application of meta-modelling and model transformation.

1 Introduction

MDA exploits the emergence of a class of tools, which support model translation and allow meta-model manipulation.

Meta-models are models of the formalism used to build models. They define the various kinds of contained model elements and the way they are arranged, related and constrained. The process of developing a model results in the creation of instances of the model elements defined in the meta-model – the meta-model is "populated" with instance data.

Model transformation is the process of converting a model expressed in one formalism to another model of the same system expressed using a different formalism. This can be achieved by building a meta-model of each of the source and target model representations and then defining a mapping between them. The meta-model of the source model is populated with instance data of the specific source model to be transformed. The mapping rules are applied as a set of operations invoked on the source meta-model, which results in a meta-model of the target model populated with instance data. This populated target meta-model is then used to generate the target model (or possibly the target text in the case of code generation).

[1] Teamwork is a graphical software-modelling tool originally marketed by Cadre.

This paper illustrates some practical examples of the use of meta-modelling. The meta-models described in this paper were all developed using the Kennedy Carter modelling tool known as iUML, and expressed in executable UML (xUML), which is a UML subset. If the UML is to evolve into a programming language that will eventually displace the use of 3rd Generation Programming languages[2] (3GL's), then it must become stable, precise and executable. The original UML specification was not sufficient for executable modelling. To cater for this, it had to be extended by the addition of action semantics, which were added at UML 1.5. The Action Specification Language (ASL) used in xUML is proprietary to Kennedy Carter Ltd. but it is compliant with the UML Action Semantics. The study demonstrated the effectiveness of xUML and ASL in building some fairly complex meta-models. This relied to a great extent on the fact that ASL is a language specifically designed to manipulate UML models.

2 Transforming Teamwork Models to iUML Models

The work in transforming Teamwork Models to iUML Models arose from concerns over software method and tool obsolescence, and the fact that many products were developed using the Structured Analysis (SA) Methodology adopted and fashionable in the late 1980's and early 1990's. These products and their associated software and models may have to remain in service and be subject to maintenance and significant upgrade for anything up to 50 years of service life and yet the Teamwork modelling tool is no longer supported. The intention therefore was to demonstrate that models developed in Teamwork could be automatically transformed in their entirety to profiled UML models, which could be maintained, checked and further developed using the iUML tool.

Each Teamwork element was mapped to an equivalent profiled UML element. For example a "Data Process" in SA was mapped to a "Passive Class" in UML (a class with no state machine), a "Control Process" was mapped to an "Active Class" (a class with a state machine), a "Data Flow" was mapped to an "Association" as was a "Control Flow" etc. A naming convention identified the different types of elements.

It was possible in this way to represent Data Flow Diagrams as UML Class Diagrams. The automated process that was developed enabled the layout of the UML diagram to appear the same as in the source Teamwork diagram, by carrying across the topology information stored in the Teamwork model. There were obvious cosmetic issues; e.g. a circular Teamwork process bubble now appears as a rectangular iUML class symbol. Further work would be needed to address symbol shapes and whether the association lines used to represent data flow did in fact flow through the centre of the bubble symbol.

The Teamwork to iUML tool migration process is illustrated in Fig. 1 above and consists of the following steps:

1. Build a meta-model of the old tool export format (in the case the textual notation used to store Teamwork models known as CDIF[3]).
2. Build a meta-model of the old formalism, in this case Real Time Structured Analysis (RTSA). A fragment of this meta-model is shown in Fig. 2.

[2] 3GL refers to Ada, C, C++, Java etc.

[3] CDIF (CASE data interchange format) is a proposed standard, which addresses model and method independent transfer between CASE tools.

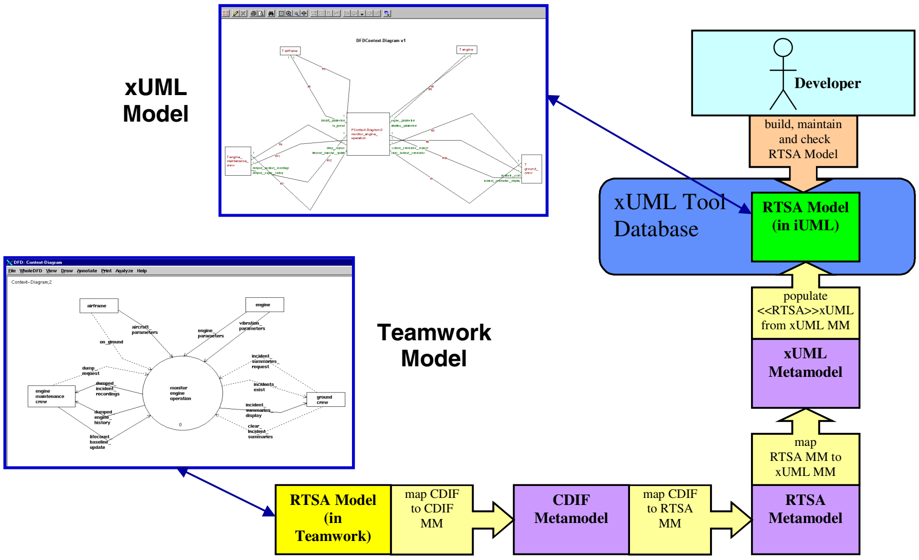


Fig. 1. Teamwork to iUML tool migration process

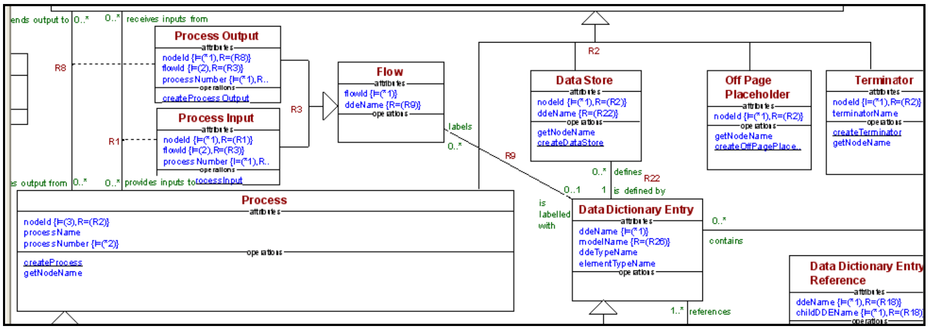


Fig. 2. Fragment of Structured Analysis Meta-Model

3. Specify the mappings from the CDIF meta-model to the RTSA meta-model, which maps a CDIF process component to a RTSA process.
4. Build a mapping from the RTSA meta-model to the UML meta-model.
5. Build a mapping from the UML meta-model to the new tool database (i.e. the iUML database).

The resulting iUML models are meta-model oriented and not diagram oriented as in Teamwork. This means that one fact is now stored in only one place in the new iUML representation of the SA model, e.g. if a store name is changed in one place, it changes it everywhere, both in the graphical representation and in the corresponding data dictionary entry. In principle these new iUML based SA models can be made executable by the addition of action specifications and executable statecharts. Full automatic code generation from these models also becomes a possibility.

The example here was one of migrating models built using the Real-Time Structured Analysis method on the Teamwork tool, originally developed by Cadre. However, the approach could potentially be used for any method and tool combination. It should be noted that in this case the modelling tool that was used to model and generate the translation tools for the transformation is also the one that is used for modelling the software applications.

3 Migration of Legacy Software Methodology to MDA Methodology

This work continued on the theme of the management of software method obsolescence with regard to legacy systems. The previous study dealt with the issue of Teamwork obsolescence, but continued to support the established SA methodology using a UML tool. A further study investigated the incremental migration of existing components of a legacy avionics system to a new methodology. The rationale was that there would be a cost-reduction benefit in the maintainability and adaptability of costly legacy software investments. The need is to avoid the systems definition ultimately residing in reams of code, maintained by engineers who have difficulty understanding the intentions of the original developers. The intention is that such systems can be migrated to assets expressed as much more easily comprehensible models, which are technology independent and from which code can be automatically generated. The models and not the code become the controlled source, and these models should be designed for reuse.

The Captor radar is fitted to the Typhoon aircraft. The radar software was developed using SA and Teamwork. Some of the existing functionality associated with the air to surface tracking was remodelled in xUML using the iUML tool. The objective set for the exercise was to determine whether Ada 83 code could be generated from the xUML tool. This code was to be integrated into the radar in place of the code that it displaced, and must match the displaced code in functionality and performance. This was successfully demonstrated.

An important theme for all the participants in the study was to understand the technology involved in building an MDA style Ada 83 Code Generator, and to establish whether safety arguments could be applied to it.

The Ada 83 code generator, developed using Kennedy Carter’s Configurable Code Generator (iCCG), is realised as a set of meta-models, where each meta-model covers a different aspect of the xUML formalism. iCCG is a product that provides a framework for developing code generators (or model compilers as they are often called) that translate xUML models into 3GL code.

The MDA specification defines a process based on model transformations, which are from the Platform Independent Model (PIM) to Platform Specific Model (PSM) to Platform Specific Implementation (PSI) as shown in Fig. 3.

The Ada code generator that was actually built for this demonstration is known as a formalism-centric code generator. It is driven at the highest level by elements of the

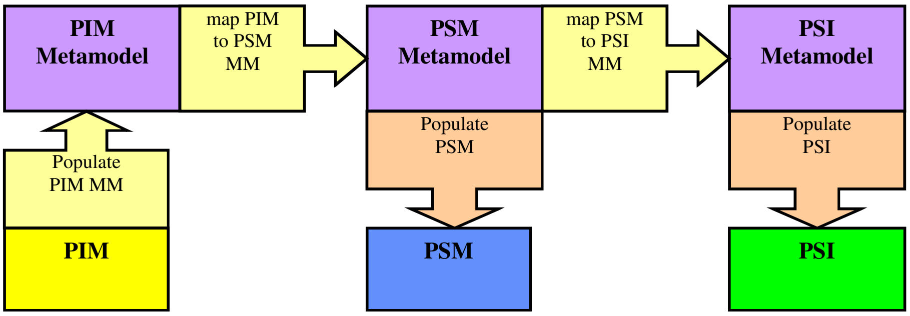


Fig. 3. The MDA Code Generation Process

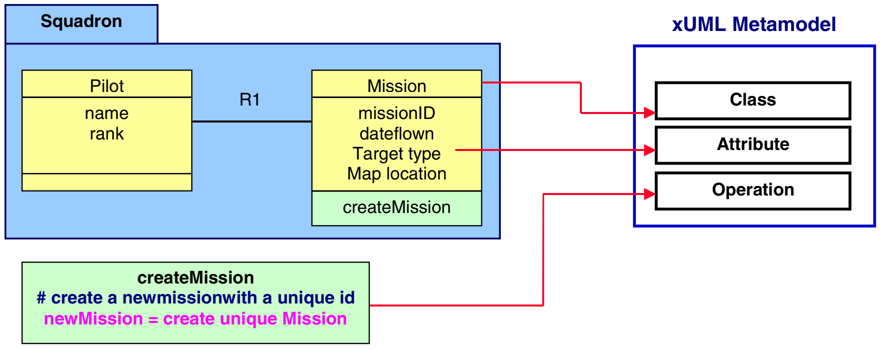
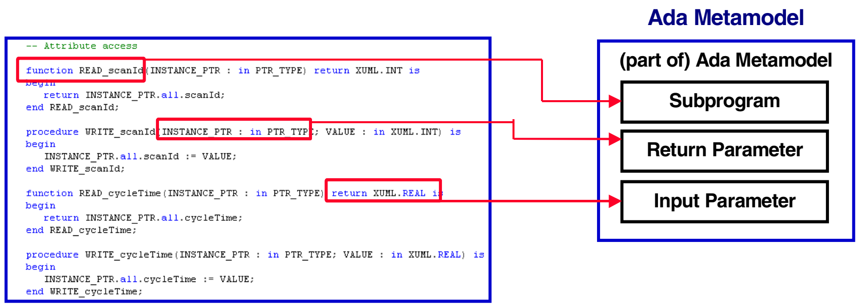


Fig. 4. Simplified Platform Independent Model and Meta-model

xUML formalism, i.e. domain, class, etc. Fig. 4 gives a simplistic indication of the subject matters in the xUML model and the xUML meta-model that constitutes the PIM. The code generator was developed therefore in two phases:

1. A set of rules was defined that described how to map every element of an xUML model to code.
2. The rules were implemented using iCCG by adding operations to the xUML and ASL meta-models, which generated the code text.

Although the Captor processor architecture is a multi-process, multi-thread architecture, only a small amount of support for multi-threading was incorporated into the code generator to meet the needs of the level of integration that was required for the demonstration. In this code generator the xUML and ASL representations were translated directly into the Ada language (in other words the PSI, which is simply represented in Fig. 5). In such a code generator, there was little modelling of the platform software architecture, i.e. the PSM in the MDA sense. However such a code generator can be extended by capturing the subject matter of the platform in additional architectural domains (or meta-models) that supplement the assembly of



```
-- Attribute access

function READ_scanId(INSTANCE_PTR : in PTR_TYPE) return XUML.INT is
begin
   return INSTANCE_PTR.all.scanId;
end READ_scanId;

procedure WRITE_scanId(INSTANCE_PTR : in PTR_TYPE; VALUE : in XUML.INT) is
begin
   INSTANCE_PTR.all.scanId := VALUE;
end WRITE_scanId;

function READ_cycleTime(INSTANCE_PTR : in PTR_TYPE) return XUML.REAL is
begin
   return INSTANCE_PTR.all.cycleTime;
end READ_cycleTime;

procedure WRITE_cycleTime(INSTANCE_PTR : in PTR_TYPE; VALUE : in XUML.REAL) is
begin
   INSTANCE_PTR.all.cycleTime := VALUE;
end WRITE_cycleTime;
```

Fig. 5. Simplified Platform Specific Implementation and Meta-model

domains that make up a formalism-centric code generator. Such a code generator is known as a platform-centric code generator. These additional domains constitute the PSM meta-model, and when populated, constitute the PSM.

For example, a code generator could be built to cater for a multi-process, multi-thread architecture by the addition of meta-models that model the architectural features of process, thread and inter-process and inter-thread communication. The analyst accomplishes the distribution of xUML models across multiple processes by using tags, which indicate to the code generator how the model is to be distributed and thus how the PIM meta-models are to be populated. For example a *data tag* is attached to domains and classes to locate object data and a *code tag* is attached to non object-scoped operations to locate code. A simplistic representation of a PSM is shown in Fig. 7, which has been referred to as a "blueprint" model and meta-model, for reasons that will become clearer in the next section.

4 Developing Integrated Modular Avionics' (IMA) Systems

The previous section described how meta-modelling was applied to building a simplistic formalism-centric code generator and indicated how this could be extended to model the platform. In the following study, the modelling of a complex computer platform architecture was a very much more pressing issue.

Traditional avionics architectures have consisted of a federation of loosely coupled embedded computers, which are supplied by a variety of manufacturers. This results in a multi-spare maintenance policy and a poor failure recovery strategy. The adoption of a modular avionics computer architecture compatible with the adoption of an open integrated modular software architecture is regarded as a major cost saving strategy. The goals of such an architecture are:

1. Technology Transparency - The underlying hardware should not have any impact on an application either during development or execution.
2. Scheduled Maintenance - The system should have inbuilt capability to operate in the presence of failures so that extended Maintenance Free Operating Periods (MFOPS) can be achieved.

3. Incremental Update - The system should be designed such that applications can be inserted/altered with minimum impact on other applications and on the supporting safety case.

The requirement is to develop and then to map all the avionics applications to a rack of Processing Modules. The applications that are active at any time during the mission can be selected as a function of the current operational mode (e.g. take-off, reconnaissance, attack etc.). In order to cater for failure conditions, it must be possible to re-map active applications to different non-failed modules (a certain amount of redundancy is catered for). These mappings are referred to as system configurations.

This requires a sophisticated operating system that can manage the dynamic reconfiguration of the system at run-time in response to operational modes and fault conditions. Such an operating system requires instructions on how it is to behave for each system state in response to reconfiguration or error handling events, and these are stored in a "runtime blueprint". The runtime blueprint includes information about the applications, the target hardware and all the possible configurations and the mapping solutions between these.

During development it is highly desirable to develop the applications independently of the hardware and then to devise a mechanism for specifying the complex mapping combinations to the target hardware. The solution that has been devised is to create three design-time blueprints that each captures respectively the necessary information about the applications, the hardware resources and the system configurations. The mapping solutions are derived from these blueprints and captured and synthesised in a fourth design-time blueprint known as the system blueprint. The run-time blueprint is a translation of information stored in the system blueprint, and its components are distributed around the various processing elements of the system.

The four development environments and how they fit with the generation of blueprints is shown in Fig. 6. The System Modelling Environment (SME) is where the overall system is modelled and results in the Configuration blueprint. The Application Development Environment (APPDE) is where the Application is developed resulting in the Application blueprint and models. The Architecture Description Environment (ARCDE) is where the

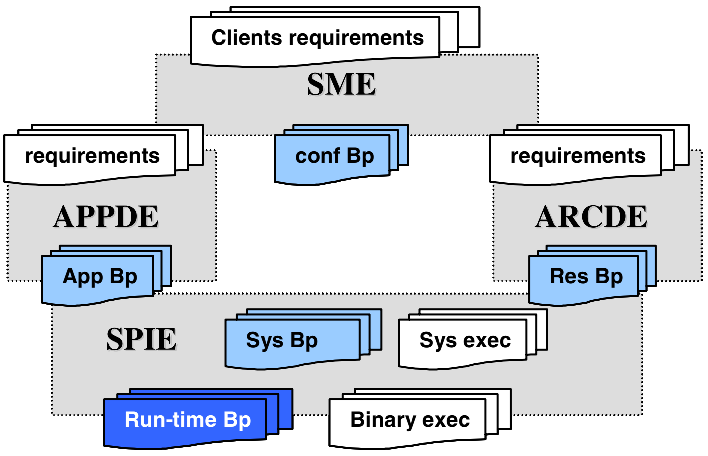


Fig. 6. Environments of the development process

hardware architecture is defined, and captured in the Resource blueprint. The System Prototyping and Integration Environment (SPIE) is where the system is prototyped and integrated resulting in the generated system software and the System blueprint.

The IMA Development Process fits closely with the concepts of MDA. The intention of the Application Domain is "to develop the applications independently of the hardware", in other words in MDA-speak, to develop a PIM.

The intention, subject matter and content of the blueprints is similar to those PSM's discussed in the previous section which are used for the construction of a platform-centric code generator. The blueprints correspond to the PSM meta-models and models illustrated in Fig. 3, their purpose is to define the software architecture and are used to map the platform independent applications to the target platform (see Fig. 7). The implication therefore is that the Blueprints have a dual role to play in the IMA software development process, not only are they used to synthesise the Run-Time Blueprint, but they can also be used in the auto-generation of the code, i.e. the PSI.

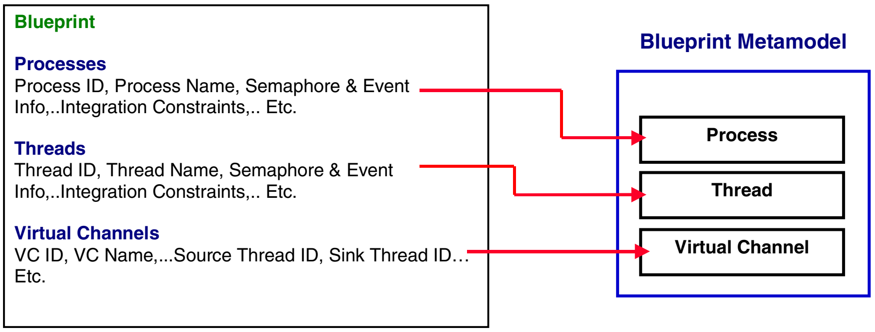

Blueprint
Processes
Process ID, Process Name, Semaphore & Event Info,..Integration Constraints,.. Etc.
Threads
Thread ID, Thread Name, Semaphore & Event Info,..Integration Constraints,.. Etc.
Virtual Channels
VC ID, VC Name,...Source Thread ID, Sink Thread ID… Etc.
Blueprint Metamodel
Process
Thread
Virtual Channel

Fig. 7. Simplified Platform Specific Model and Meta-model

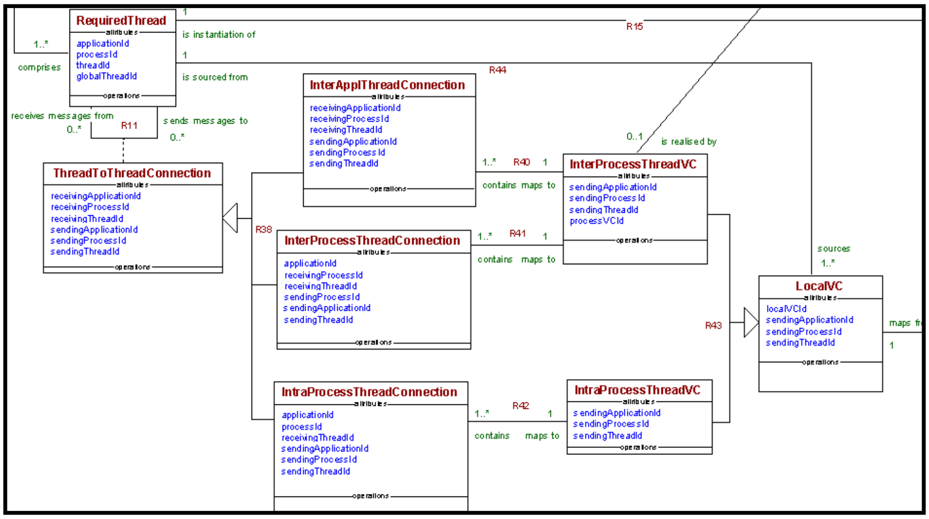


Fig. 8. A Fragment of an Application Blueprint Model

During the study, aspects of the Design Time Blueprints were modelled using a number of tools, Fig. 8 shows some of the work that was done with the iUML tool (this diagram should be regarded as purely experimental, it does not correspond to any realised blueprint definition).

5 Conclusions

This paper examined only one example of a number of tools that are now becoming available to address the OMG standard for Model Driven Architecture (MDA). This paper illustrated some practical examples that have demonstrated that xUML (which is a UML subset), made executable by an Action Specification Language (ASL) can be used to develop sophisticated meta-models, which can in turn be used for model transformation. Some tools use a 3GL as the UML action language, but this does present some problems with regard to building meta-models. One example is chained navigation i.e. given an instance or set of instances of a class, find by navigating a chain of associations an associated instance or set of instances. This can be expressed quite succinctly in a shorthand notation constituting a single line of ASL. This single line in turn can translate into 50 or more lines of C++ or Ada code[4] in order to execute. Such navigation chains occur all the time when gathering data from instantiated meta-models in order to perform model transformation. Clearly it is too clumsy to have to write this amount of 3GL code explicitly, for every navigation.

On the other hand the Kennedy Carter ASL lacks features of a standard 3GL that are rather desirable when designing software applications. For example, it does not have access to a maths library, which makes it necessary to revert to the native 3GL code of the target platform in order to invoke maths functions such as trig functions or matrix manipulations etc. Standards are required, and the OMG have issued requests for proposals (RFP's – the requirements document that initiates the standard setting process) for executable UML and concrete action language syntax.

The study demonstrated the potential for preserving investment in existing models built using tools that are no longer supported through model transformation. Model transformation also offers the potential for automatic data transfer between disparate specialist modelling tools used in the development tool chain.

The study also demonstrated that a common approach and formalism can be applied to the development of both models of software applications and to the meta-models used to transform them. The ability to generate not only the components of the target system, but components of the development tool chain, provides scope for model translation and offers "executable specifications" that can be tested early and mapped reliably onto the target, leading to greater levels of dependability.

The traditional software approach makes a rather fuzzy distinction between analysis, which is the definition of what the system is to do, and design, which is how the software is to be implemented. The boundary where one finishes and the other starts, is equally difficult to define. No such distinction is made in this approach. The modelling method and formalism to develop the application as a PIM, is exactly the

[4] The reason that this happens is because of the need to iterate and find matches over sets of data.

same as that for modelling the target platform as the PSM. The difference is a clean subject-matter separation between application behaviour and platform specific implementation technologies.

The software architecture devised for Integrated Modular Avionics is probably the ultimate PIM in terms of sophistication and complexity. Meta-modelling presents a solution to modelling the Blueprints as the PIM's that define the software architecture. These in turn offer the dual role of (a) the repositories for capturing the mapping solutions used to build the run-time blueprint, and (b) the mechanisms to build the mappings required to auto-generate the implementation code as the PSI.

Acknowledgements

The help of Chris Raistrick of Kennedy Carter Ltd. in the preparation of this paper and the participation of the industry, academic and tool vendor partners in the joint research work referred to in this paper is hereby acknowledged.

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. UML Distilled Applying the Standard Object Modeling Language. Martin Folwer, Kendall Scott ISBN 0-201-32563-2
6. Model Driven Architecture with Executable UML. Chris Raistrick et al. ISBN 0-521-53771
7. Model Driven Architecture – An Industry Perspective. Chris Raistrick, Tony Bloomfield Architecting Dependable Systems II. ISBN 3-540-23168-4

Using Domain Driven Development for Monitoring Distributed Systems

Rainer Burgstaller[2], Egon Wuchner[1], Ludger Fiege[1], Michael Becker[1], and Thomas Fritz[3]

[1] Siemens AG, Corporate Technology, D-81730 Munich, Germany
[2] TU Darmstadt, Darmstadt, Germany
[3] LMU Munich, Munich, Germany

Abstract. Domain Driven Development (DDD) is one key to conquering the complexity of large-scale software systems. The use of domain-specific models raises the level of abstraction in programming, and helps modularizing and automating the development process. However, in pilot projects using MDx approaches we have experienced the need for more tool support on model level, in particular for monitoring, debugging, and testing. In this paper, we investigate monitoring and feedback mechanisms that make models reflecting the current state of the running system. Additionally, we describe a prototypical implementation, which is a starting point for further work on enabling debugging and testing on model level.

1 Introduction

Large software systems imply a number of severe problems that make their development and maintenance very complex: huge code bases mixing application logic with infrastructure code, missing or not up-to-date documentation, and the need to create new and customized versions in short time intervals. New technologies provide easy, often ad hoc, solutions for engineering distributed systems, but they hardly raise the level of abstraction. Engineers have to deal with lots of source code to incorporate changes on the business level.

Model driven software development (MDSD) promises to mitigate these problems by raising the level of abstraction and automating some development steps. Domain Driven Development (DDD) approaches work with descriptions, i.e., models, expressed in terms of the respective application domains. The domain-oriented view eases communication with domain experts, limits the involved amount of details, and modularizes the development process. To some extent, DDD is an extension of OMG's take on MDSD with its Model Driven Architecture (MDA) [4].

DDD automatically transforms one or more input models and finally generates code in the last step of the process. Application programmers can then extend the code base to refine the application-specific logic. The generated code is not changed so that the generation step can be reiterated when models change.

However, so far this approach does not cover the whole software engineering lifecycle. The transformation process is currently one-way, focusing on transforming models into code, while support for debugging and testing is rarely available on the model level. Domain experts participating in the development process want to monitor, debug, and test running instances. But unlike traditional software engineers, they should stay on the model level, not being forced to work on code level.

In this paper, we investigate mechanisms that feed runtime status information back into the model to synchronize it with the running instance. Such mechanisms are the basis for monitoring, debugging, and testing on model level. We focus on how monitoring can be achieved in domain driven development of distributed systems. We present a prototype application that helped us to investigate the main problems.

The remainder of this paper is organized as follows. In Section 2, we present the example application and the monitoring functionality we wanted to achieve. This is followed by our prototype implementation in Section 3. Finally, Section 4 summarizes the paper and gives an outlook on future work.

2 Domain Driven Development

2.1 Example Application

Our example application is an inventory tracking system (ITS) [5]. The system is used to monitor and control the flow of goods and assets in warehouses. Typically, an ITS consists of three types of systems, operating on three different levels. At the top level—the *operation* level—operate systems that are responsible for planning, scheduling and supervising the progress of all business-level operations. Next, at the intermediate level—the *process control* level—resides the warehouse management system (WMS) responsible for the accurate and timely execution of all activities planned and scheduled at the operation level. Finally, at the bottom level—the *entity* level—operate physical devices like forklifts, conveyer belts, storage racks and others, performing the operations requested by the WMS. Goods delivered at a warehouse gate must be moved to appropriate storage locations, and requested goods must be looked up and conveyed to the delivery gate. System operations are monitored by the warehouse management system. It visualizes ongoing work, offers a management console for manual interactions, checks for alarm conditions, and alerts the operator if necessary.

2.2 Applying DDD

The main objectives when developing the warehouse management system were to involve domain experts in all stages of design and operation, to exploit existing components for material flow and the control subsystem, and in particular to automate the software process as far as possible. Therefore, we have applied a domain driven development approach. Domain experts create a model of

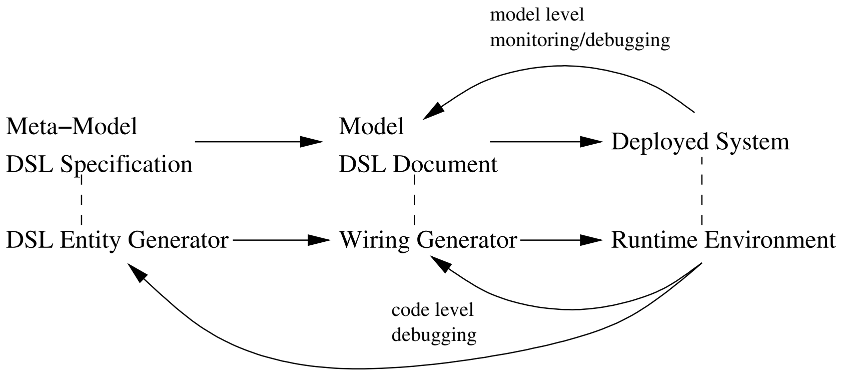


Fig. 1. Domain and code level development

the warehouse that defines its specific layout and additional rules of operation. Domain specific languages (DSLs) keep domain experts away from code level representations and often provide graphical models of the system that are well understood by these experts.

The main benefit of the DDD approach is that domain experts participate in the definition of the DSL itself, i.e., of the warehouse meta-model, and they also use the DSL to describe individual warehouse instances. They work on a domain-oriented view of the warehouse and do not need to care about code level artifacts, or component or deployment diagrams. In DDD, domain level and code level engineering should go in parallel throughout the whole software lifecycle (see also Figure 1).

The definition of the warehouse DSL includes generators to create code (skeletons) for the DSL entities. Models defined with the DSL are complemented with generators that wire the code for a specific warehouse instance. As for all model driven approaches, the generation step should be repeatable so that domain experts may update the model without destroying code level contributions. However the backward direction is hardly supported, so far.

2.3 Need for Monitoring

The goal of DDD is to involve domain experts also in optimizing layout and operation, and in adapting the system to changing requirements. This requires involving them in defining QoS, alerting states, checking and correcting constraints, and in debugging. Obviously, the next step in the DDD process would be to allow for model level monitoring of the deployed system. We need such a monitoring as part of our application anyway, and in general it is the basis for further steps in the engineering process, namely for debugging and testing purposes.

A generic approach for monitoring arbitrary models of a given DSL requires at least:

- *Tool support:* Assuming we use a graphical DSL, it would be useful to have information about the deployed system fed back into the same tool used for creating the model.
- *The possibility to specify monitored artifacts:* Typically, not all deployed artifacts and not all details are of interest. We need a mechanism to specify which details should be monitored, or under which conditions the information should be fed back into the tool.
- *The ability to select, filter, and transmit information:* We need communication mechanisms that are able to filter and convey the detected data from the deployed system into (one or more) monitoring tools. To support large distributed systems, we probably need event notification mechanisms as underlying means of communication [1].

3 Realization

After having motivated our approach, we will now explain its realization. On the one hand, we will give an overview of the transformation process in which the different code-related files are generated from the model. On the other hand, we will shortly describe the employed event-notification mechanism.

To realize the feedback component, we used Ice [3] and some of its services, e.g., IceStorm (a publish-subscribe service that decouples clients and servers). Ice is an object-oriented middleware platform that allows you to build distributed applications with minimal effort. Like Corba's Interface Description Language (IDL), Ice uses an abstract description language (called Slice — Specification Language for Ice) to define the client-server contract independent of a specific programming language. For a good overview of Ice and a comparison to Corba see [2].

Model-Code Transformation. We used Visio as our *Modeling Tool*, as it offers meta-modeling and domain-modeling capabilities, is easy to extend for our generation purpose and has a lot of other user-friendly features [5]. Our generator transforming meta- and domain-models is implemented as a plug-in for Visio written in C#.

First, one has to specify a meta-model by defining the master shapes in Visio, including the definition of properties, constraints and connection points. Starting from this meta-model, the generator iterates over all defined master shapes and generates Slice files (see step one in Figure 2). Each Slice file represents one master shape and contains its interface definition. These slice files are then used in a second step to generate the corresponding proxy and skeleton code in C# by the Slice-To-C#-Compiler of Ice (relevant for the client-server communication)—see step two.

In step three, the generator again iterates over all master shapes of the meta-model. Thereby, it creates a C# class for each master shape that inherits from the according skeleton class. These newly generated classes act as servants and, as the relevant methods are overridden and feedback functionality is added to

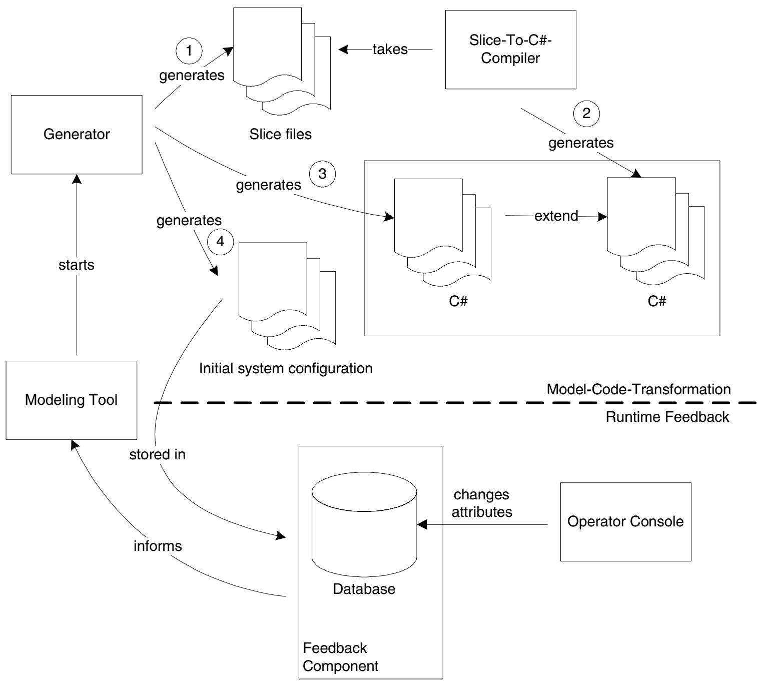


Fig. 2. Monitoring approach. The arrows show the communication between the components.

them, they already contain the notification calls in every method that changes attribute values.

Once the meta-modeling and transformation is done, one can use Visio to draw domain models containing a certain number of different shapes and to specify attribute values of the model elements. Out of this domain model, the generator now creates an initial system configuration that is used later on at start-up of the system in order to initialize the systems' objects. Only the first time the system is initialized using this generated configuration, later on, the relevant information on the object states is retrieved from the database.

Runtime Feedback. Now that the system is running, one can monitor dynamic changes of the domain model with our feedback mechanism. Therefore, the modeling tool (Visio) is connected with the running system using the Feedback Component. Now, each time an object changes its state (triggered by an algorithm or via the operator console), the feedback component notifies the modeling tool about the change by firing an event (including the changed data). This notification handling has been generated into the servant classes as described above.

In the Visio plug-in developed for our prototype the user can choose whether he wants to have monitoring support or not. If requirements change (so that, for instance, monitoring support is not needed any more), a simple regeneration and recompilation is necessary. Another possibility would be to unconnect the modeling tool but then a certain overhead for the signaling mechanism remains.

4 Concluding Remarks

Currently, model driven software development does not cover all stages of software engineering. Current approaches support only one-way transformations from the model into the code and miss feedback mechanisms enabling debugging and testing on the model level. Our approach is a first step towards a complete modeling approach that comprises model-code transformation as well as dynamic feedback. We have demonstrated an example showing the usefulness of monitoring facilities on the model level. Furthermore, we presented our feedback approach, an event-based mechanism that tracks dynamic changes in the system. Feeding runtime information back into the model is the basis for monitoring, debugging, and testing.

Future work could comprise a more fine-grained feedback mechanism. Currently, all runtime changes of model elements are tracked. However, it might be preferable to specify the elements of interest on the model level, maybe even in a new layer of abstraction.

References

1. Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.
2. Michi Henning. A new approach to object-oriented middleware. *IEEE Internet Computing*, 8(1):66–75, 2004.
3. Michi Henning and Mark Spruiell. Distributed programming with Ice, 2005.
4. Jishnu Mukerji and Joaquin Miller. MDA Guide, v1.0.1. omg/03-06-01, June 2003.
5. Andrey Nechypurenko, Tao Lu, Gan Deng, Douglas C. Schmidt, and Aniruddha Gokhale. Applying MDA and component middleware to large-scale distributed systems: A case study. In *First European Workshop on Model Driven Architecture with Emphasis on Industrial Application*, Enschede, The Netherlands, March 2004.

Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code[⋆]

Sven Burmester[⋆⋆], Holger Giese, and Wilhelm Schäfer

Software Engineering Group, University of Paderborn,
Warburger Str. 100, D-33098 Paderborn, Germany
{burmi, hg, wilhelm}@uni-paderborn.de

Abstract. The model-driven software development for hard real-time systems promotes the usage of the platform independent model as major design artifact. It is used to develop the software logic at a high level of abstraction and enables analysis like for example model checking of critical model properties. Ideally, starting with the platform independent model, the platform specific model serves only as an intermediate artifact which is derived automatically, and will finally result in a set of threads whose implementations guarantee the behavior, specified in the platform independent model. However, the current MDA approaches and tools for hard real-time software do not provide this ideal: While some of the MDA approaches could in principle support this vision, most approaches simply do not support an appropriate specification of time constraints in the platform independent model which have to be respected in the platform specific model or in the code. This is also true for UML models and UML State Machines in particular. Our approach overcomes those UML specific limitations by firstly proposing a syntactic extension and semantic definition of UML State Machines which provides enough details to synthesize an appropriate platform specific model that can be mapped to code for hard real-time systems automatically. Secondly, a new partitioning algorithm is outlined, which calculates an appropriate mapping onto a platform specific model by means of real-time threads with their scheduling parameters which can be straight forward transformed to code for the hard real-time system.

1 Introduction

The current practice when building software components with hard real-time constraints is characterized by the following step-wise partially manual process: (1) *Specification:* The software is specified on a high abstraction level (if at all), then (2) *Partitioning:* The software is partitioned into concurrent threads with appropriate periods to make it run on a real-time operating system (usually without adequate analysis), (3) *Implementation:* The software is implemented (often manually, which makes implementation

[⋆] This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and founded by the Deutsche Forschungsgemeinschaft.

[⋆⋆] Supported by the International Graduate School of Dynamic Intelligent Systems. University of Paderborn.

faults very likely), (4) *Analysis:* It is verified that the software fulfills all real-time constraints in its environment (testing as employed in practice is usually not sufficient for complex software to guarantee the absence of timing errors). If the real-time constraints do not hold, partitioning, implementation and analysis have to be repeated. Repeating this cycle a number of times is usually very costly but often unavoidable.

Consequently, there is an increasing demand to extend model-driven architecture (MDA) [1,2] to design software for embedded hard real-time systems. When using MDA for such systems, the developer would have to specify the so called *Platform Independent Model (PIM)* which describes the system behavior including the real-time constraints which must be met. Ideally, a tool would then automatically partition the specification and map it to the *Platform Specific Model (PSM)*, based on a *Platform Model (PM)* that provides details about the target platform. The PSM describes the active objects and their scheduling parameters which are required to implement the system behavior, specified by the PIM. In the next step, the PSM would be compiled automatically into the platform specific implementation which guarantees a correct implementation of the PIM's semantics. The implementation would guarantee the real-time constraints by construction and thus, no verification of the real-time constraints is required. This would make the above mentioned manual steps *(3) Implementation* and *(4) Analysis* unnecessary. Such guarantees for the derived implementation further permit to analyze other required properties or to reveal faults or inconsistent real-time constraints using the platform independent model rather than considering the much more complex code (e.g. by model checking [3,4,5]).

One reason, why the indicated iterative manual process is followed in practice instead of the MDA approach, is that currently, there exists no support to automatically map a PIM to a PSM that is appropriate for real-time systems. The UML [6] can be considered as the standard to model complex software systems even in the real-time domain [7,8,9,10]. Consequently, we propose in the paper an approach to realize the above outlined vision with UML, even though UML has not been originally designed to support real-time systems and a semantically correct implementation for standard UML State Machines is due to the underlying zero execution time semantics not possible. Although ROOM [11] has finally found its way into UML 2.0, the required support for real-time behavior modeling is still not available, as the ROOM concepts focus on architectural design and do not address the real-time behavior of the operational model at all.

Another thread of development is the *UML Profile for Schedulability, Performance, and Time* [8]. The profile defines general resource and time models which are used to describe the real-time specific attributes of the modeling elements such as scheduling parameters or quality of service (QoS) characteristics. However, it remains an open question in the UML profile how all required details are determined. In a scenario where the developer derives these details from a high-level (platform independent) model and maps them on technical concepts such as threads and periods manually, we still have the problem that this mapping results in an iterative manual process of testing and adjusting the model until the real-time constraints are met. Nevertheless, the profile defines an appropriate level of abstraction to be used as PSM. This PSM can be later used for further model analysis (e.g. scheduling analysis) and code generation.

To provide an appropriate PIM, we first propose a syntactic extension of UML State Machines and a related semantic definition. By enriching the model with deadline information (besides others), our extension provides enough details in the PIM to synthesize a PSM and finally code for hard real-time systems. We provide the PM by a description of worst case execution times (WCETs) of local side-effects and of the code fragments that will be used in the automatically generated implementation of the state machine.

For the automatic derivation of the PSM from our extended notion of State Machines, we developed an algorithm for automatic partitioning and for automatic derivation of scheduling parameters. The algorithm takes CPU time sharing on a single micro processor into account. An automatic implementation usually leads to less faults than a manual implementation. The automatic partitioning respects the deadlines from the PIM and the WCETs from the PM.

Therefore, the algorithm for automatic *Partitioning* and *Implementation* guarantees that all real-time requirements are met, which makes the *Analysis* unnecessary and avoids the costly iterative process of *Partitioning*, *Implementation* and *Analysis*. If the algorithm fails to provide a partitioning, the model is not realizable.

The next section presents our approach for platform independent modeling of hard real-time systems and relates it to standard UML models. Section 3 describes the platform model and the component's deployment. Section 4 shows in detail how to derive a platform specific model and finally code. Section 5 discusses current approaches of specification techniques for embedded systems with hard real-time constraints and their limitations. Finally, Section 6 draws a conclusion and sketches current and future work.

2 Platform Independent Models

In this section, we first describe how to specify the system's structure. Then, we discuss in detail how to specify the behavior of components of embedded real-time systems with UML and with our approach. Finally, we present our analysis methods.

2.1 Structure Modeling

Embedded real-time systems consist of a complex architecture of components (cf. Figure 5). In [3], we have presented an approach how to specify the architecture and complex real-time communication between the components by UML component diagrams and patterns respectively. Our approach further permits to verify the component's interconnection by means of compositional model checking assuming that each single component behaves as specified. How the single component's real-time behavior is specified and how it is correct implemented automatically is described in the remainder of this paper.

2.2 Behavior Modeling

We use an example from the RailCab research project[1] as our running example. The vision of the RailCab project is a rail system where autonomous operating shuttles apply

[1] http://www-nbp.upb.de/en/index.html

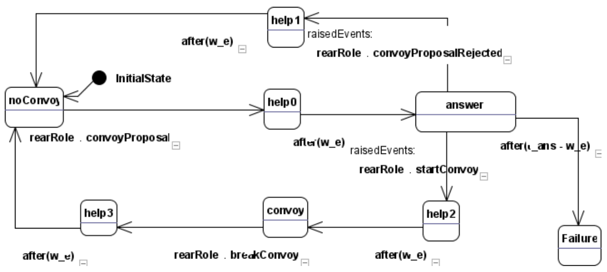


Fig. 1. UML approach to model the shuttle coordination

the linear drive technology used in the Transrapid, but travel on the existing passive track system of the standard railway. One particular problem, which has been previously described in [3], is to reduce the energy consumption due to air resistance by forming convoys whenever possible. Such convoys are created on-demand and require small distances between the shuttles in order to achieve significant economies.

Building convoys changes the shuttles' behavior (e.g. the way of accelerating and braking). Thus, It must be guaranteed that all involved shuttles of a convoy switch to convoy mode in an appropriate and predictable amount of time which results in a number of hard real-time constraints.

After receiving a convoyProposal message, that denotes a request to build a convoy, we demand for the communication that the shuttle answers within the time t_{ans} with rejection (message convoyProposalRejected) or with acceptance (startConvoy).

In a first attempt to describe this coordination with a UML State Machine, the state machine would switch to an intermediate state when receiving convoyProposal. This intermediate state would be left via a transition labeled with after(t_{ans}) to switch to a Failure state if no answer was sent during this time.

The semantics of such a model assumes the transitions to be fired within zero-time, but this is not realizable in an implementation in real life systems due to three reasons: (i) Consuming or raising events or executing side-effects consumes time. (ii) An implementation of a state machine requires a task which periodically checks if transitions are triggered. As only positive, *non-zero* periods are realizable, this leads to a further delay. (iii) If other processes are executed on the processor, further delays occur due to scheduling.

One possibility to model time consumption of raising events or executing side-effects is the use of the after-construct as shown in Figure 1. In order to respect the worst case execution time w_e for consuming or raising events the help-states are introduced. They are entered when an event is consumed or fired and left after w_e (to simplify the example we assume that consuming and raising of events consumes the same amount of time).[2]

Such a description models correctly that the actions consume time (cf. (i) above), but still consist of transitions that react infinitely fast (cf. (ii)) and do not respect scheduling

[2] Note when regarding Figure 1 that we denote the sending of a message msg to target tgt by tgt.msg. Receiving from receiver rcv is denoted by rcv.msg.

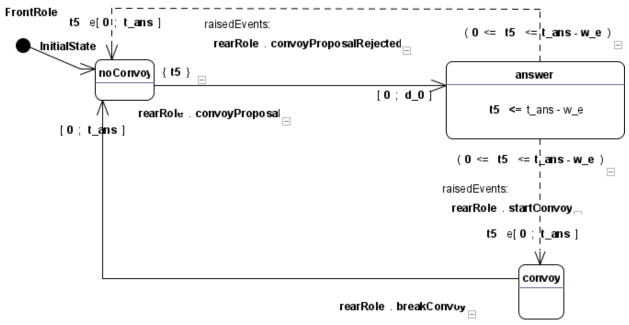


Fig. 2. Real-Time Statechart

delays (cf. (iii)). Further, the after-construct is used in 2 different ways: after($t_{ans} - w_e$) specifies the point in time when the according transition has to fire (as proposed by the UML). Contrary to this, after(w_e) is used to model the progress of time while raising an event.

The example illustrates, that UML State Machines are not practical for our demands and that there is need for a realistic model that supports the specification of hard real-time constraints like WCETs and upper bounds for reaction times.

The abstraction of *zero execution time*, employed in UML State Machines, is often interpreted to mean *fast enough*. Thus, to specify *how fast* they have to react, we propose to specify deadlines for each required side-effect. Thus, in our *Real-Time Statechart* model [12,13], which is an extension of the UML State Machine model, transitions are not assumed to fire *infinitely fast*, which is unrealistic on real physical devices (especially when considering the execution of the actions attached to the transitions), but it is possible to specify deadlines for each transition which in turn determine what *fast enough* really is.

These time constants specify a relative point in time defining the minimum time (always 0 in this example, see Figure 2) and the maximum time (d_0, t_{ans}) until the firing of the transition has to be finished. These points in time are either absolute in relation to the point in time when the transition has been triggered (e.g. the transition from noConvoy to answer) or relative to a clock. In the example, the deadline $t_5 \in [0; t_{ans}]$ of the transition from answer to convoy is relative to the clock t_5. t_5 is reset to zero when switching to state answer (indicated by $\{t_5\}$ similar to the notion in timed automata [14,15]). The clock is reset at the point in time when the transition is triggered. The deadlines avoid to use extra or help states (as in Figure 1) and thus enable to construct a less complex model in terms of the number of states.

Further, we enhance the model –similar to timed automata– by time invariants defining the point in time when the state has to be left via a transition. The state answer is only valid as long $t_5 \leq t_{ans} - w_e$ holds. To trigger transitions dependent on a specific point in time, time guards are specified (e.g. $0 \leq t_5 \leq t_{ans} - w_e$).

Transitions are triggered when the time guard becomes true, the associated event is available and a guard, consisting of a boolean expression over different variables or methods, is also true. We distinguish between *urgent* transitions (visualized by solid arrows) firing immediately when they are triggered and *non-urgent* transitions (visualized by dashed arrows). The latter ones may be delayed when the time specifications of the model still allow a later firing [14]. Urgent transitions are similar to eager transitions in [16] and non-urgent transitions are similar to delayable or lazy transitions in [16]. They are used to model different possible alternatives in the communication protocol. This introduced non-determinism is resolved in Figure 3 showing the whole shuttle behavior.

The after-construct is mapped to a time guard and a time invariant and thus gets a semantic definition which makes it possible to generate code from this definition. Although the use of multiple clocks requires more effort than using the after-construct, it has the advantage that the points in time, when transitions are triggered, cannot only be defined relative to the point of entrance of the current state, but also relative to the point of triggering of any previously fired transition or the point of entrance or exit of any previously entered state, because clock-resets can be associated even to the exit()- and entry()- methods of the states.

The form of the time guards is limited to $\wedge_{t_i \in C}(a_i \leq t_i \leq b_i)$, $a_i \in \mathbb{N}$, $b_i \in \mathbb{N} \cup \{\infty\}$, where C is the set of clocks. The form of time invariants is limited to $\wedge_{t_i \in C}(t_i \leq T_i)$, $T_i \in \mathbb{N} \cup \{\infty\}$. In our experience, this limitation, i.e. the exclusions of arbitrary logic expressions and arithmetic operations on different clock times, does not hamper the modeling of realistic systems and makes it easier for the model developer to build intuitive models rather than very complicated ones.

The semantic definition of Real-Time Statecharts does not have the usual macrostep and run-to-completion semantics of UML State Machines, because the zero execution time for intermediate steps is not realistic in our application domain. Many actions have significant WCETs. Run-To-Completion semantics would not allow an immediate reaction to any newly raised external event. We define our semantics formally, as given in [13] by a mapping of Real-Time Statecharts to a subset of an extended version of hierarchical timed automata as defined in [17]. Such a semantics has already been employed successfully in a similar domain [18] for the un-timed case. In order to still be able to describe the required local synchronization between multiple orthogonal states of a single Real-Time Statechart within a single step, synchronous communication via synchronization-events and -channels, similar to the mechanism described in [14], is also supported.

Apart from the above mentioned extensions, which are partly adapted from timed automata, features from UML State Machines like hierarchy, parallelism and history as well as entry()-, exit()- and do()-operations for states are, of course, provided further on. While a specific state is active, its do()-operation is executed periodically. The user may specify a time interval for this period. Actions are not limited to integer assignments (like in timed automata), but can be complex method calls in the object-oriented model. The WCETs are respected in the PM (see Section 3).

Figure 3 shows the whole shuttle behavior, consisting of three orthogonal states.[3] The upper orthogonal state realizes the described part of the communication protocol. The lower orthogonal state realizes the opposite part. The orthogonal state in the middle

[3] Note that in our CASE tool *Fujaba* (www.fujaba.de) the dashed lines between orthogonal states are not visualized.

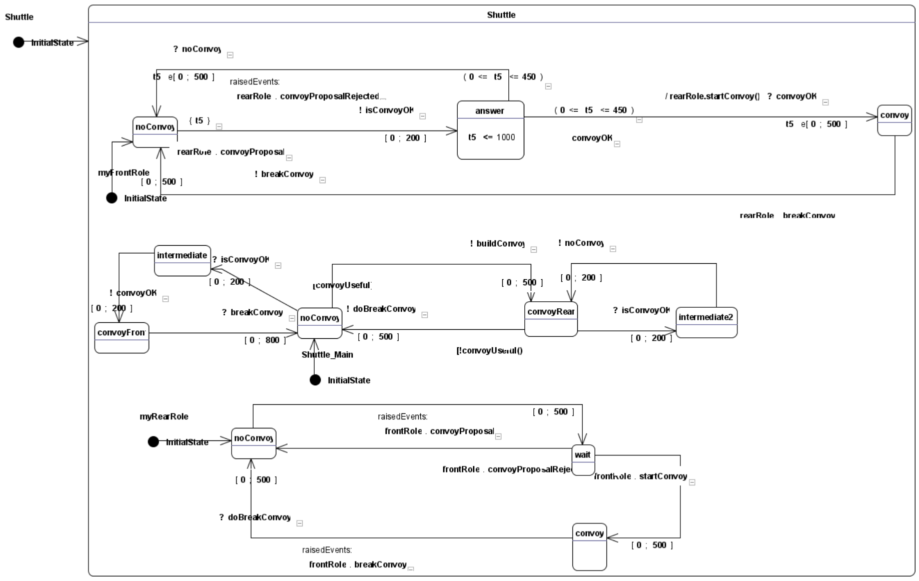


Fig. 3. Behavior of a shuttle component

synchronizes both roles. It initiates the building and the breaking of the convoy. In this simplified example, convoys consisting of maximal two shuttles are build.

Real-Time Statecharts combine the advantages of UML State Machines and of timed automata and extend them by additional annotations. These annotations enable to generate the PSM and finally code for real-time platforms on the one hand and offer constructs to model complex temporal behavior on the other hand. The main differences to UML State Machines are, that they (1) support to model the time consumption of transition execution and (2) have a realistic semantic definition based on timed automata mirroring appropriately the application domain.

2.3 Model Analysis

Generating a PSM, consisting of active objects and deadlines, that guarantee the real-time constraints as specified in the model is of course only possible, when the model does not contain any conflicts between the declarative elements such as time guards and time invariants. A possible conflict is for example when multiple real-time constraints are contradicting and thus no behavior exists which fulfills them (time-stopping deadlock).

To exclude such conflicts, the full state space of a Real-Time Statechart model has to be checked in the general case. Due to the well-defined semantics of Real-Time Statecharts [13], which map their behavior to hierarchical timed automata as employed in the model checker UPPAAL [19,14], we first map them to hierarchical timed automata and then feed them into the vanilla extension of UPPAAL [19] which flattens them in an additional preprocessing step. Then, this flat timed automata model is checked with UPPAAL for the absence of time-stopping deadlocks or other required properties expressed with a restricted temporal logic.

When a time-stopping deadlock has been found, we have to conclude that the final state of the delivered error trace contains a conflict. Pinpointing the root source of the problem is a complex problem which remains to be done manually.

While model checking the PIM provides a high cost solution in the general case, we can do much better for specific failure classes where the complex dependencies which result from the synchronization between orthogonal states are ignored when the deployment and thus the platform model is known.

Imagine, as one example for such a static analysis, a state with (a part of) an invariant $t_i \leq T_i$, which is the source of a set of leaving transitions which all have a time guard of the form $T_i + x \leq t_i$, $x > 0$. It is obvious, that once entered, this state will never be left again and a time-stopping deadlocks occurs.

Our additional static analysis algorithms employed upfront detect such temporal inconsistencies at low costs. Due to the incompleteness of the analysis, it is a supplement to model checking but cannot, of course, replace it to detect all inconsistencies in the general case. The pessimistic analysis further indicates whether model checking is required at all or whether the much simpler static checking for temporal inconsistencies has been sufficient.

3 Platform Models and Deployment

In order to generate the PSM, WCETs are required for all actions (side-effects, entry(), exit(), and do()- operations) and for the elementary instructions that build the code fragments realizing the Real-Time Statechart behavior (e.g. checking guards, raising events, etc.).

3.1 Deployment

As the WCETs are platform-dependent, we first deploy our components (whose behavior is each specified by a Real-Time Statechart) by a UML deployment diagram. In such a deployment diagram, we assign the component instances of our systems to dedicated nodes and the cross node links to available network connections in form of busses or direct communication links. Given such an assignment, we can further look into the specific characteristics of the different nodes as described in the platform model.

3.2 Platform Models

In the platform model, the relevant characteristics such as CPU type, operating system, etc. are described. Therefore, available techniques to determine these single WCET values as described in [20] can be employed. They allow to annotate these values to the platform specific view of the behavioral elements such as methods and elementary instructions. The WCETs of the code fragments of a Real-Time Statechart can then be determined by summing up the execution times of the elementary instructions and more complex methods.

3.3 Model Analysis

To analyze the resulting model with platform specific annotations, we extend our timed automata model for model checking as well as our static analysis technique such that it also reflects the WCET behavior of the side effects of the transitions.

A temporal inconsistency can, for example, occur, if a time guard, a time invariant, and a WCET are in contradiction. One case is given by a time guard which can trigger a transition at a point in time, when the execution of the action will not be possible, because the time invariant of the target state may have been exceeded after execution of the action. Consider, for example, a transition with a time guard $t_0 \leq 10$ and an action with a WCET of 4 leading to a state with the invariant $t_0 \leq 12$. If this transition is triggered, for example at $t_0 = 10$, the target state is entered in the worst case at $t_0 = 14$, which violates the time invariant.

Such problems can be detected using model checking. In addition our static analysis algorithms can be upfront detect some of these temporal inconsistencies at low costs as in the case of the PIM analysis.

4 Synthesizing Platform Specific Models and Code

After modeling and analyzing the PIM with components and Real-Time Statecharts and specifying the platform specific WCET information in the PM and the deployment, we have to map the components and links to active objects and to network and communication links to come up with the final platform specific model. In our case the PSM can be described by the UML Profile for Schedulability, Performance, and Time [8], as it allows the specification of priorities, periods, and deadlines for active objects. We use it as platform *specific* model, as these values, which we derive automatically from the platform *independent* model, are different for different platforms.

When building real-time systems, cost saving requires to minimize hardware costs. Consequently, the number of processors and their power is restricted. Thus our mapping algorithm is designed for single processor systems, whereby all branches of the orthogonal states are mapped to one single processor. In case the system consists of multiple components, deployed on different processors, every component executes on exactly one processor. Thus, the mapping algorithm can then be applied, too.

One periodic thread ensures that the Real-Time Statechart reacts *fast enough* to meet all time restrictions. The thread's period defines *how fast* the Real-Time Statechart reacts. Its determination, that considers the specified attributes (deadlines, etc.) as well as the externally determined WCETs, builds the main part of Section 4.1.

As every Real-Time Statechart is implemented as exactly one active object which will be implemented as periodic thread (and possibly multiple aperiodic threads), the number of concurrently running threads can become large when plenty of Real-Time Statecharts are executed on the same processor. If this is the case (e.g., for UML models with a large number of active objects or components), we propose to combine multiple Real-Time Statecharts into a single one using orthogonal states to optimize the result of the partitioning. Using such a grouping, an unacceptable overhead due to a large number of threads is avoided and we still resolve the partitioning and scheduling problem by employing the proposed code generation algorithm.

4.1 Partitioning

As mentioned above, a Real-Time Statechart is mapped to at least one periodic thread, checking for triggered transitions in every period – the so called *main thread*. This thread checks all transitions which can be triggered from the beginning of its last period until and during the duration of the current period. The checking has to be started that early, because the check in the last period may have just missed a transition which could have been triggered. It was missed because the check happened just before the event occurred or its time guard was evaluated to true.

A transition is triggered, if the following four conditions hold: (1) The transition is defined for the current state, (2) the event has occurred in the time interval between the beginning of the last period (of the main thread) and the current point in time (Note that an event, that cannot be consumed immediately, is queued), (3) the time guard is evaluated to true during or after the event happens, (4) the guard is evaluated to true during or after the event happens. The worst case time needed for the whole check (depending on the current state) is denoted by $w_{\mathrm{trig}}(s)$, where s is the current state.

After determining all triggered transitions and the points in time when they became activated, the first triggered transition is fired. Then clocks are reset and actions are executed.

If the action has such a short WCET, such that there is still enough execution time left within the period, it will be executed by the main thread. As it is possible to specify complex actions, their WCETs often do not fit into the main thread. If they are executed within the main thread nevertheless, its execution time would become greater than its period and deadline. Apart from this problem, the main thread would not be able to check and – if needed – fire other transitions for the time the action is executed, although this is required in the case of orthogonal states. Due to these problems, the firing of such transitions is rolled out into a new started aperiodic thread, running concurrently to the main thread. Thus, orthogonal states are not implemented by multiple concurrent running periodic threads, but by exactly one periodic thread and multiple concurrent running aperiodic threads. Among other things, this facilitates the efficient implementation of synchronization.

The still remaining problem is to determine the main thread's period. On the one hand, it needs to be short enough, such that the recognition of triggered transitions happens early enough to guarantee that the actions are executed before their deadlines expire. On the other hand, it should be as long as possible to execute as many transitions as possible within the main thread and thus to minimize resource utilization, because an additional aperiodic thread consumes time and memory. Respecting these conditions, the annotations and restrictions in the Statechart specification as well as the times $w_{\mathrm{trig}}(s)$, w_{start} and w_{end} give limits for the duration time of one period. w_{start} and w_{end} denote the duration for starting and terminating an aperiodic thread. We determine the period for a target system, scheduled by a priority scheduler [21]. When deriving equations to determine the period from the specification, several cases need to be distinguished.

Figure 4a shows the first case when $w_{\mathrm{trig}}(s)$ and the action to be executed (WCET is denoted with w_a) fits into the periodic thread. The execution has to guarantee that the action is executed before its deadline expires, i.e. the period is short enough to execute

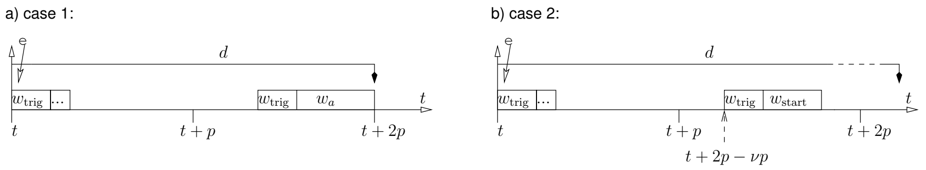


Fig. 4. Determining the period

the action before the deadline d. The worst case in terms of a delay between triggering a transition and executing its corresponding action is the following: The main thread begins execution at time t – the beginning of a first period and just misses a transition which is triggered. As we apply priority scheduling, that transition is only checked again and fired at the end of the execution of the next period ($t+p$ until $t+2p$) such that it just fits into this period (p denotes the duration time of one period). Then $d \geq 2p$ must hold in order to be sure that the action is executed before the deadline expires. Respecting the so called *utilization factor* $\nu \in (0;\ 1]$, defining that a Real-Time Statechart shall not gain more than ν percentage of the processor load, obviously $w_{\mathrm{trig}}(s) + w_a \leq p\nu$ must hold for cases where the processor load is shared.

This results in the inequality $p^1_{\mathrm{min}} := (w_{\mathrm{trig}}(s) + w_a)/\nu \leq p \leq \frac{d}{2} =: p^1_{\mathrm{max}}$ determining minimum and maximum values for p in case of executing an action within the main thread (case 1).

A more complex situation occurs when $w_{\mathrm{trig}}(s) + w_a \leq p\nu$ does not hold and the action needs to be rolled out to an aperiodic thread, like shown in Figure 4b. Although the start of the aperiodic thread shortens the necessary execution time of the periodic main thread to $w_{\mathrm{trig}}(s) + w_{\mathrm{start}}$, we still compute an upper bound which minimizes rollouts. In any case, the computation time within every period, the main thread gets, is $p\nu$. Even, when this time is not enough to execute the action, the periodic thread is started at least $p\nu$ time units before the end of it's period, cf. Figure 4b. In this case, the computation time is not used completely by the periodic thread. The remaining time is already used by the started aperiodic thread. Consider the (trivial) case $\nu = 1$: The delay between triggering the transition and executing the action is given by $p + w_{\mathrm{trig}} + w_{\mathrm{start}}$. Then, the action is executed and the aperiodic thread terminates. Thus the delay, the execution and the termination have to fit into the deadline: $d \geq p + w_{\mathrm{trig}} + w_{\mathrm{start}} + w_a + w_{\mathrm{end}} \Leftrightarrow p \leq d - w_{\mathrm{trig}} - w_{\mathrm{start}} - w_a - w_{\mathrm{end}}$.

While the aperiodic thread is executing, the periodic one still runs (with a shorter execution time w'_g) and preempts the aperiodic one once within a period. A detailed analysis (which is given in [22]), respecting these preemptions and $\nu \in (0,\ 1]$ leads to the inequality 1, that uses the substitutions $\alpha = (\nu w_a + \nu w_{\mathrm{end}} + w_{\mathrm{trig}} + w_{\mathrm{start}})/\nu^2$, $\beta = w'_{trig}/\nu$, $\varphi = w_a - w_{\mathrm{trig}} - w_{\mathrm{start}}/\nu$.

$$p \leq d - \alpha - \beta \left\lceil \frac{\nu p - \varphi}{p + \beta} \right\rceil \tag{1}$$

Applying a numerical algorithm leads to the solutions in the form $p \leq p^2_{\mathrm{max}}$. Considering the necessary execution time $w_{\mathrm{trig}}(s) + w_{\mathrm{start}} \leq \nu p$ leads to another inequality

$p^2_{\min} := (w_{\text{trig}}(s) + w_{\text{start}})/\nu \leq p \leq p^2_{\max}$ determining minimum and maximum values for p in case of executing an action, that is rolled out (case 2).

The period has to fit either the first or the second inequality. As a Statechart usually consists of multiple transitions, a period is chosen, that fits at least one equation for *every* transition. For the case that a state is entered and a leaving transition becomes triggered immediately, two more inequalities arise, because besides the action the do-operation needs to be executed, too. Further, the period has to fit either the third or the forth equation, too. Analyzing the specified Real-Time Statechart leads to a system of inequalities consisting of four times as much inequalities as transitions occur in the Statechart. Thus, choosing the period is a combinatorial problem, that is solved automatically by a numerical method. If multiple solutions exist, the period for the main thread will be the longest one possible. After determining the period, it is fixed which actions need to be rolled out to aperiodic threads.

4.2 Platform Specific Model

Figure 5 depicts the structural view of the PIM and the according generated PSM. The Shuttle component is transformed to the active class MainThread, realizing the periodic main thread and to the class Shuttle, realizing the logic of the component. The determined period, which is equal to its deadline, is annotated as proposed by the Profile for Performance, Schedulability, and Time. The priority is determined according to the deadline monotonic approach [21]: The thread with the shortest deadline, achieves the highest priority. Note that the deadlines of the aperiodic threads, that execute long side-effects are specified in the PIM.

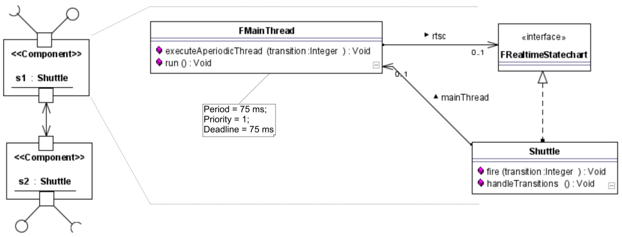


Fig. 5. PIM and PSM of the shuttle system

4.3 Model Analysis

When implementing applications for embedded real-time systems, resource restrictions need to be taken into account. Memory and computation time are usually the restricted resources in embedded systems. As the structure of our models is static and thus there is no need for dynamic instantiation, the required memory can be derived straight forward and is fixed after partitioning. To check if sufficient computational power is available, especially when multiple Real-Time Statecharts or other processes have to be executed on one microprocessor, scheduling analysis is performed. Note that even when the sum of all processes' utilization factors is less or equal 100%, schedulability cannot be guaranteed without adequate analysis [21].

In order to speed up scheduling analysis, we first use Liu and Layland test [23] to make a rough estimate and apply Lehoczky's, Sha's, and Ding's analysis algorithm [24] only if needed. If the set of *all* threads is not schedulable, we exploit the knowledge about the possible concurrently executed threads which can be derived from the structure of the Real-Time Statechart. For example, aperiodic threads, initiated by firing transitions, that are executed sequentially, will never be executed concurrently. All combinations of threads, that can possibly run concurrently are determined and it is sufficient to check the schedulability for all these combinations.

4.4 Code Generation

Using the automatically generated PSM, the mapping to a real-time target platform, supporting priority scheduling, is straight-forward. Currently, we support the generation of Real-Time Java [25] and C++ for an appropriate real-time operating system. In this generation step, active objects are mapped to real-time threads. The result of this mapping can be imported into our CASE Tool Fujaba by its reengineering capabilities.

5 Related Work

Currently available approaches for the specification and implementation of hard real-time systems have the following disadvantages: Either, they offer the required higher level modeling concepts, but provide no partitioning and code generation concepts which ensure the specified hard real-time behavior of the model, or they support code generation which guarantees timing behavior, but are already platform specific models.

In [9], Rational Rose models are extended with information needed for scheduling and partitioning in form of periods and action WCETs. This information is then used to distribute the components automatically to multiple processors and to guarantee schedulability. This approach is, however, rather limited as synchronization within the components (usually described by Statecharts) is not supported.

Hierarchical timed automata [17], which are a hierarchial extensions of timed automata [14,15], provide most of the powerful modeling concepts of Statecharts. A mapping to multiple parallel running flat timed automata permits to verify the model by using the model checker UPPAAL [14]. In [26], locations of a flat UPPAAL automaton are associated with tasks inclusive WCETs and deadlines. This extension enriches the model with the information required for code generation and a prototype synthesizing C-Code has been implemented. As the code generation approach is restricted to flat automata, it does not take the additional syntactical constructs of hierarchical timed automata into account. The code generation scheme is not really sufficient for hard real-time systems, as it does not take into account the delays that occur when transitions are fired, arguing that these delays are small compared with the WCETs.

Modecharts [27] are another high-level form of state transition systems for the specification of real-time systems. Actions are executed only while residing in states and not when firing transitions. The model respects that actions require time and thus they are associated with deadlines or –if needed– with periods. Timing constraints like deadlines and trigger conditions are specified just relative to the current state's (mode's) point of entry and not relative to preceding states. [28] describes code generation for the target language ESTEREL, but the generated implementation regards only the timing intervals, triggering the transitions and not the deadlines or periods.

SAE AADL (Society for Automotive Engineers Architecture Analysis & Design Language) [29], successor of MetaH,[4] specifies a system on the PSM level. A SAE AADL model consists of multiple Threads, annotated by a priority and a frequency and can therefore be mapped to code automatically. Tool support for modeling is currently restricted to text based editors.

The application framework VERTAF [30] and the automata model presented in [31] specify the required real-time constraints and thus enable an automatic implementation. These approaches are not applicable for complex systems, as their models are rather restricted: [31] applies just a flat automata model, [30] specifies active object on the implementation level.

Currently available CASE tools Rhapsody, Rational Rose/RT, Statemate, TelelogicTau, and Artisan Real-time Studio Professional for UML State Machines can only generate code from the logical behavior, while an appropriate mapping onto threads and scheduling parameters in form of the synthesis of a platform specific model remains to be determined in a manual process. To the best of our knowledge all existing UML CASE tools also fail to close the gap between high level models and the automatic implementation of hard real-time systems.[5] In contrast, the presented approach supports the automatic synthesis of the PSM from a given PIM and PM.

6 Conclusion and Future Work

Our approach, consisting of components and Real-Time Statecharts, permits to specify complex real-time systems following UML notations and the MDA approach at the PIM level. This platform independent description can then be mapped automatically to a platform specific model, provided that a target platform description in form of annotations describing real physical behavior (WCETs) are given. The PSM describes real-time threads, which are of general nature and not bound to a specific programming language or RTOS environment. Thus, an implementation can be realized in any programming language that provides real-time priority scheduling. Different analysis methods are applied on the different levels to achieve correct models.

Right now, the open-source UML CASE tool Fujaba supports modeling with UML components and Real-Time Statecharts including model checking and code generation for Real-Time Java and C++ from UML components and Real-Time Statecharts. We are currently extending Fujaba to explicit visualize the generated PSMs and to permit manual adjustments like adding other threads to the system's nodes. In this context, we prove if the standard UML Profile for Schedulability, Performance, and Time

[4] www.htc.honeywell.com/metah

[5] We refer to [7] for a judgment that Rhapsody (www.ilogix.com) and Rose/RT (http://www-306.ibm.com/software/rational/) only support soft real-time system development. We further evaluated Artisan Real-time Studio Professional (www.artisansw.com), Statemate (www.ilogix.com), Rational Rose/RT, and Telelogic Tau G2 Developer (www.telelogic.com) on our own.

is sufficient or if extensions like for example the HIDOORS Profile [32,33] are required. Automatic grouping of Real-Time Statecharts and modular code generation for deployment-time grouping is planned future work.

References

1. Allen, P., ed.: The OMG's Model Driven Architecture. Volume XII of Component Development Strategies, The Monthly Newsletter from the Cutter Information Corp. on Managing and Developing Component-Based Systems. (2002)
2. Object Management Group: MDA Guide Version 1.0. (2003) Document omg/2003-05-01.
3. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the Compositional Verification of Real-Time UML Designs. In: Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland, ACM Press (2003)
4. Burmester, S., Giese, H., Hirsch, M., Schilling, D.: Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In: Proceedings of the International Workshop on Specification and vaildation of UML models for Real Time and embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004. (2004)
5. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (2000)
6. Object Management Group: UML 2.0 Superstructure Specification. (2004) Document: ptc/04-10-02 (convenience document).
7. Bichler, L., Radermacher, A., Schrr, A.: Evaluation uml extensions for modeling realtime systems. In: Proc. on the 2002 IEEE Workshop on Object-oriented Realtime-dependable Systems WORDS'02, San Diego, USA, IEEE Computer Society Press (2002) 271–278
8. Object Management Group: UML Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/02-03-02 (2002)
9. Gu, Z., Kodase, S., Wang, S., Shin, K.G.: A Model-Based Approach to System-Level Dependency and Real-Time Analysis of Embedded Software. In: The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, Canada. (2003)
10. Masse, J., Kim, S., Hong, S.: Tool Set Implementation for Scenario-based Multithreading of UML-RT Models and Experimental Validation. In: The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, Canada. (2003)
11. Selic, B., Gullekson, G., Ward, P.: Real-Time Object-Oriented Modeling. John Wiley & Sons, Inc. (1994)
12. Burmester, S., Giese, H., Tichy, M.: Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML. In Assmann, U., Rensink, A., Aksit, M., eds.: Model Driven Architecture: Foundations and Applications. Volume 3599 of Lecture Notes in Computer Science (LNCS)., Springer Verlag (2005) 47–61
13. Giese, H., Burmester, S.: Real-Time Statechart Semantics. TechReport tr-ri-03-239, University of Paderborn (2003)
14. Larsen, K., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. Springer International Journal of Software Tools for Technology **1** (1997)
15. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic Model Checking for Real-Time Systems. In: Proc. of IEEE Symposium on Logic in Computer Science. (1992)
16. Bornot, S., Sifakis, J., Tripakis, S.: Modeling Urgency in Timed Systems. In Roever, W.P.d., Langmaack, H., Pnueli, A., eds.: Compositionality: The Significant Difference; COMPOS '97, Bad Malente, Germany, September 8 - 12, 1997. Volume 1536 of Lecture Notes in Computer Science., Springer Verlag (1998) 103–129

17. David, A., Möller, M., Yi, W.: Formal Verification of UML Statecharts with Real-Time Extensions. In Kutsche, R.D., Weber, H., eds.: 5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002), April 2002, Grenoble, France. Volume 2306 of LNCS., Springer (2002) 218–232
18. Köhler, H., Nickel, U., Niere, J., Zündorf, A.: Integrating UML Diagrams for Production Control Systems. In: Proc. of the 22^{nd} International Conference on Software Engineering (ICSE), Limerick, Irland, ACM Press (2000) 241–251
19. David, A., Moeller, M.: From HUPPAAL to UPPAAL: A translation from hierarchical timed automata to flat timed automata. In: TechReport BRICS RS-01-11, Department of Computer Science, University of Aarhus. (2001)
20. Erpenbach, E.: Compilation, Worst-Case Execution Times and Scheduability Analysis of Statechart Models. Ph.D.-thesis, University of Paderborn, Department of Mathematics and Computer Science (2000)
21. Buttazzo, G.C.: Hard Real Time Computing Systems: Predictable Scheduling Algorithms and Applications. Kluwer international series in engineering and computer science : Real-time systems. Kluwer Academic Publishers (1997)
22. Burmester, S.: Generierung von Java Real-Time Code fr zeitbehaftete UML Modelle. Master's thesis, University of Paderborn, Paderborn, Germany (2002)
23. Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM **20** (1973)
24. Lehoczky, J., Sha, L., Ding, Y.: The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In: Proceedings of the 10th Real-Time Systems Symposium. (1989)
25. Bollella, G., Brosgol, B., Furr, S., Hardin, S., Dibble, P., Gosling, J., Turnbull, M.: The Real-Time Specification for Java^{TM}. Addison-Wesley (2000)
26. Amnell, T., David, A., Fersman, E., Pettersson, M.O.M.P., Yi, W.: Tools for Real-Time UML: Formal Verification and Code Synthesis. In: Workshop on Specification, Implementation and Validation of Object-oriented Embedded Systems (SIVOES'2001). (2001)
27. Jahanian, F., Mok, A.: Modechart: A Specification Language for Real-Time Systems. In: IEEE Transactions on Software Engineering, Vol. 20. (1994)
28. Puchol, C., Mok, A., Stuart, D.: Compiling Modechart Specifications. In: 16th IEEE Real-Time Systems Symposium (RTSS '95), Pisa, Italy. (1995)
29. Feiler, P.H., Gluch, D.P., Hudak, J.J., Lewis, B.A.: Embedded Systems Architecture Analysis Using SAE AADL. Technical Report CMU/SEI-2004-TN-005, Carnegie Mellon University (2004)
30. Hsiung, P.A., Su, F.S., Gao, C.H., Cheng, S.Y., Chang, Y.M.: Verifiable Embedded Real-Time Application Framework. In: Seventh Real-Time Technology and Applications Symposium (RTAS '01), Taipei, Taiwan. (2001)
31. Saksena, M., Karvelas, P., Wang, Y.: Automatic Synthesis of Multi-Tasking Implementations from Real-Time Object-Oriented Models. In: The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Newport Beach, California (2000)
32. Richard-Foy, M., Hunt, J.J.: The HIDOORS Profile: Applying the Scheduling, Performance and Time Profile to Realtime Java Development. In Amann, U., ed.: Proc. of Model Driven Architecture: Foundations and Applications (MDAFA 2004), Linköping, Sweden. (2004)
33. Meunier, J.N., Lippert, F., Jadhav, R., Harding, N.: MDA and Real-Time Java: The HIDOORS project. In Akehurst, D., ed.: Proc. of Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations (EWMDA-2 2004), Canterbury, England. (2004) 89–95

Model-Driven Performance Analysis of UML Design Models Based on Stochastic Process Algebra

Naoshi Tabuchi, Naoto Sato, and Hiroaki Nakamura

IBM Research

Abstract. The popular model-driven development (MDD) methodology strongly promotes a model-based approach to modular system development, which often implies as an integral part automatic transformation of UML design components into executable forms. When using MDD for verifying performance-related system specifications, UML designs annotated with these specifications in some profile language need to be transformed to stochastic (Markovian) models or timed simulation models. However, most of the previous efforts have focused on transformations of (variants of) UML state machine models and/or transformations to stochastic Petri net models, which lead to two problems: Relying (solely) on state machine models often restricts design flexibility (designers instead prefer choosing diagrams on a case-by-case basis), and graph-oriented Petri net models complicate the modular transformations of UML models.

To resolve these problems, we propose stochastic performance analysis of a UML design defined in different sorts of diagrams, including not only state machines but also activity diagrams with temporal annotation in UML-SPT, which are transformed into stochastic process algebraic forms. To our knowledge, this is the first attempt to define stochastic process algebraic semantics for the UML AD with UML-SPT annotations. Unlike the related efforts, ours will facilitate verification in early development stages, in which consultants and architects can benefit from modeling in a natural manner and modular component-based development, thanks to the inherent compositionality of process algebra. Further, to guarantee the validity of the transformation, we have proved the equivalence of our semantics with the stochastic Petri net semantics of UML AD.

We have developed a prototype implementation of this performance analysis mechanism, and shown that realistic design examples, defined in different sorts of UML diagrams, can successfully be transformed into those that provide various performance metrics.

1 Introduction

The popular model-driven development (MDD) methodology strongly promotes a model-based approach to modular system development [16], which often implies as an integral part automatic code generation from design components in Unified Modeling Language (UML) [14] to analyze system properties. Since the analysis of performance-related aspects of systems often becomes particularly crucial, it turns out that UML models need to reflect those performance-related features. To this end, OMG has standardized a profile language, called UML-SPT [13], which provides a rich set of vocabulary to describe performance-related aspects of systems. Relying on this or similar (but non-standard) notations, several efforts have been devoted to transforming UML models with performance annotations to stochastic models or timed simulation models for their performance analysis [12,20,5].

However, most of these efforts have focused on transformations of (variants of) UML state machine models and/or transformations to stochastic Petri net models, which lead to the following problems: (1) The behavioral aspects of a system would be modeled as a set of *different types of diagrams*, including not only state machine diagrams but *activity diagrams* as well, though the latter has not so far been taken into consideration. (2) According to the UML standard, invocations of component operations need to be defined as *CallOperation* actions in activities. This implies that it is virtually impossible to define component compsitions without relying on activity diagrams (unless resorting to some non-standard notations). (3) Graph-oriented Petri net models complicate modular transformations of UML models. The Petri net formalism, in its original form, defines a system of components as a single flat graph of places and transitions, and does not provide any systematic support for combining those that are transformed from different components.

To resolve these problems, we propose a model transformation of UML 2.0 design models, which may include activity and state machine diagrams annotated with performance-related values in UML-SPT, into stochastic process algebraic forms. By employing this approach, system architects are liberated from state-machine-centric behavioral modeling and granted freedom to choose diagram types. Once UML models with performance annotations are defined, they are transformed in a unified manner into stochastic algebraic forms for performance analysis. Thanks to the inherent compositionality of the process algebra, the transformations can be applied to the components of a system separatedly, so that external references in a component are left unresolved through the component transformation and are taken care of by using the process composition mechanism of the process algebra. In short, our approach extends the flexibility of UML-based performance analysis by allowing behavioral modeling in both activity and state machine diagrams, and enables modular model transformations by mapping components of a UML model to processes in the process algebra.

Specifically, our technical contributions are two fold: First, we have defined the formal semantics of a core part of the UML 2.0 Activity Diagram metamodel. To our knowledge, this is the first attempt to define stochastic process algebraic semantics for UML AD. Further, to guarantee the validity of the transformation, we have proved the equivalence of our semantics with an existing stochastic Petri net semantics of UML AD. Through preliminary evaluation, we have applied our transformation mechanism to a large embedded system example and have shown its feasibility and strong potential.

The rest of the paper is organized as follows. The following section briefly summarizes key concepts for model-driven performance analysis, including UML/UML-SPT, stochastic process algebras, and stochastic Petri nets. Succeedingly, Section 3 introduces a set of rules to transform UML AD, with annotation in UML-SPT, to a stochastic process algebraic language called IMC [3], and Section 4 shows the validity of the transformation by proving the equivalence between our process algebraic semantics and the existing Petri net semantics of UML AD. The last sections report results of preliminary evaluation, provide a comparison with related work, and discuss possible future directions.

2 Stochastic Performance Analysis

2.1 Modeling in UML 2.0 with UML-SPT Annotations

Elements of a UML model are divided into two categories, static structures and dynamic behaviors. Static structures define the components in the target system, their functionalities and composition. Dynamic behaviors describe how these functionalities act and interact. In UML 2.0, behaviors are defined in terms of the *Behavior* metaclass, which is further divided into three sub-metaclasses: *Activity*, *StateMachine*, and *Interaction*. Components with dynamic behaviors are represented as instances of *BehavioredClassifier*, which has a *classifierBehavior* and a set of *Operations*. Figure 1 shows a simplified view of the metamodel of *Behavior* and *BehavioredClassifier*. (The gray-colored metaclasses are not explicitly mentioned in this paper.) The behavior of each component is determined by its *classifierBehavior* and *methods* of operations, therefore it is in general a mixture of *Activity*, *StateMachine* and *Interaction*. To derive executable code from such models, we must transform different sorts of behavioral models in an integrated manner.

As our running example, let us consider the following SimpleCopier system, consisting of three components, namely Scanner, Printer, and Buffer (FIFO with a capacity of 3 pages). As a system, SimpleCopier scans incoming data, subsequently stores the data for each page temporarily in the buffer, and eventually prints out the pages. Notice that Scanner and Printer run in parallel, so the total performance of the system depends on the throughputs of both Scanner and Printer. Figure 3 shows the static structure of SimpleCopier and the behavioral description of its components. The *classifierBehavior* of Scanner is defined as an activity diagram, which infinitely repeats the

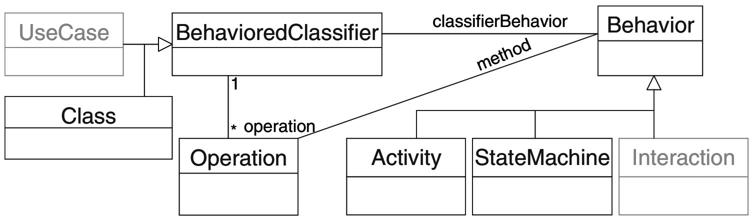


Fig. 1. Metamodel of *BehavioredClassifier* and *Behavior*

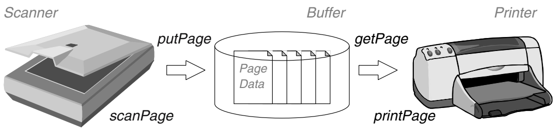


Fig. 2. SimpleCopier that consists of Scanner, Printer, and Buffer

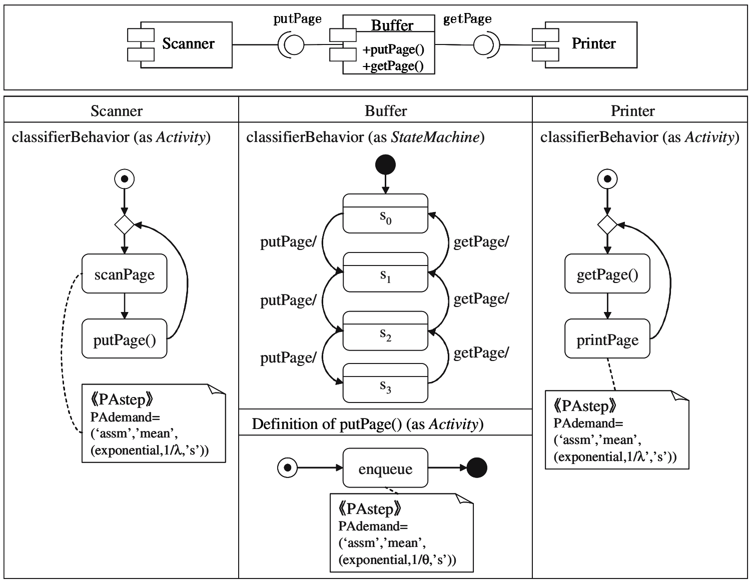

putPage
getPage
Scanner
Buffer
+putPage()
+getPage()
Printer
Scanner
Buffer
Printer
classifierBehavior (as *Activity*)
classifierBehavior (as *StateMachine*)
classifierBehavior (as *Activity*)
scanPage
putPage()
《PAstep》 PAdemand= ('assm','mean', (exponential,1/λ,'s'))
s_0
s_1
s_2
s_3
putPage/
getPage/
Definition of putPage() (as *Activity*)
enqueue
《PAstep》 PAdemand= ('assm','mean', (exponential,1/θ,'s'))
getPage()
printPage
《PAstep》 PAdemand= ('assm','mean', (exponential,1/λ','s'))

Fig. 3. UML Design Model for the SimpleCopier Components

scanPage action and the putPage() call to the Buffer. Printer is defined in a similar manner. Meanwhile, the *classifierBehavior* of Buffer is a state machine with 5 states (the initial state and $s_0 \ldots s_3$ in Figure3.) Labels of transitions represent *CallTriggers* which wait for the specified operation to be called from outside putPage(). Hence, it turns out the definition of Buffer includes both state machines and activities.

For stochastic performance analysis of a concurrent system like SimpleCopier, we need to define the stochastic behavior of the system in addition to its UML model. More specifically, the timed actions in each UML activity diagram should carry stochastic information. In our case, we employ notations of another UML standard, called the UML Profile for Schedulability, Performance, and Time [13] or UML-SPT for short, to specify temporal delays for actions. For example, scanPage of Scanner is annotated with an UML-SPT definition, ⟨⟨PAstep⟩⟩ PAdemand = ('assm','mean',('exponential',$1/\lambda$,'ms')), which states that the execution of scanPage takes $1/\lambda$ milliseconds on average and the probability that execution completes within t milliseconds is equal to $1 - \exp(-\lambda t)$ (i.e. the exponential distribution). As for those actions without annotations, we suppose they complete instantaneously without delays.

2.2 Stochastic Process Algebra

Stochastic process algebras are stochastic extensions of process algebras, of which the semantics are defined on a basis of the mathematical foundation of the Markov-chain theory. Along the principle of the process algebra, they model each concurrent system as communicating state-transition processes that are composed using built-in algebraic operators. In addition, these languages allow defining processes with stochastic behavior, that is, each transition of a process may associate its own random *delay* that determines how long to stay in the current state before the transition. Although the mathematical characterization of the random delays may vary, most of the stochastic process algebras, such as PEPA [10], TIPP [9], and IMC [3], concentrate on those with negative-exponential distributions due to their simplicity. For our discussion, we choose IMC as a representative one of these languages, of which the syntax is defined as follows.

$$P ::= \emptyset \mid (r).P \mid \alpha.P \mid P+Q \mid P|_L Q \mid X \mid \mu X.P$$

The constant symbol $\emptyset$ denotes a nil process that does nothing but terminates. $(r).P$ and $\alpha.P$ behave like P after a (negative-exponentially distributed) duration specified with $r \in \mathbb{R}$ or executing the immediate action α, respectively. Note that α ranges over $\mathit{Act} = \Sigma \cup \widehat{\Sigma} \cup \{\tau\}$, where Σ is the set of alphabets, $\widehat{\Sigma} = \{\widehat{\alpha} \mid \alpha \in \Sigma\}$ is the set of the communication counterparts of the actions in Σ, and τ denotes an internal action that is not observable from the outside. $P + Q$ offers a non-deterministic or a timed probabilistic choice between P and Q, depending on their prefix actions. When they are prefixed respectively with α_1 and α_2 ($\alpha_1, \alpha_2 \in A$), one of the actions is chosen non-deterministically, otherwise the processes prefixed with r_1 and r_2 ($r_1, r_2 \in \mathbb{R}$) have the likehood of being chosen with probabilities of $r_1/(r_1+r_2)$ and $r_2/(r_1+r_2)$, respectively. $P|_L Q$ performs P and Q in parallel, under the interleaving semantics, with synchronizations upon encountering actions in $L \subset \mathit{Act} \setminus \{\tau\}$. $\mu X.P$ defines a recursive process such that $P = P[\mu X.P/X]$, which can be referenced as X in the following process definition. As examples of IMC process definitions, we attach below those for the SimpleCopier components, P_{Scanner}, P_{Printer}, and P_{Buffer}.

$$\begin{aligned}
P_{\mathit{Scanner}} &\triangleq \mu X.\,((\lambda).\mathit{putPage}.X)\\
P_{\mathit{Printer}} &\triangleq \mu X.\,\big(\mathit{getPage}.(\lambda').X\big)\\
P_{\mathit{Buffer}} &\triangleq \mu X.(\widehat{\mathit{putPage}}.(\theta).\mu Y.(\widehat{\mathit{getPage}}.(\theta).X + \widehat{\mathit{putPage}}.(\theta).\mu Z.(\widehat{\mathit{getPage}}.(\theta).Y + \widehat{\mathit{putPage}}.(\theta).\widehat{\mathit{getPage}}.(\theta).Z)))
\end{aligned}$$

2.3 Generalized Stochastic Petri Net (GSPN)

The Generalized Stochastic Petri Net (GSPN) formalism [11] provides another basis for modeling concurrent systems with stochastic behavior.

As a GSPN, each concurrent system is defined as (P, T, F, R, M_0), in which a set of *places* P and a set of *transitions* T, along with a set of *flow* relations $F \subseteq (P \times T) \cup (T \times P)$, determine the graph structure of the net, while a rate function $R : T \to \mathbb{R} \cup \{\infty\}$ associates with each transition a rate value, and a marking $M_0 \subseteq P$ specifies the places where the tokens initially reside. The semantics of a GSPN is almost identical with that of regular Petri nets, except that each transition of a GSPN may have a negative-exponential duration for its delay, *or* alternatively may occur immediately (with zero delay), depending on its rate value.

To define the semantics of UML 1.5 Activity Diagram based on the GSPN formalism, Pablo et al. [12] extended the semantic domain to Labeled GSPN (LGSPN) [6], an extension of GSPN, and defined a transformation of UML AD to LGSPN. Formally, LGSPN is defined as follows.

Definition 1 (Labeled Generalized Stochastic Petri Net [12]). *A labeled generalized stochastic Petri net is a 6-tuple of the form* (P, T, F, R, M_0, L)*, where* P*,* T*,* F*,* R*, and* M_0 *respectively denote places, transitions, flow relations, a rate function and an initial*

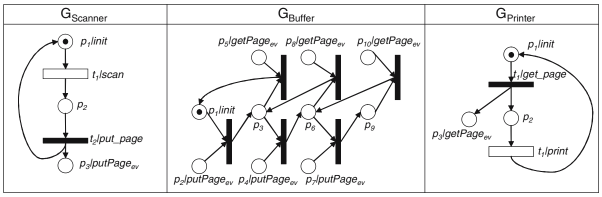


Fig. 4. LGSPN Representations of the SimpleCopier Components

marking in the same way with GSPN, and $L : T \cup P \to Lab$ *is a labeling function that associates with each place/transition a symbolic label in Lab.*

As a direct consequence of this extension, we can easily define composition (referred to as *superposition* in the literature) for LGSPNs. Specifically, given two LGSPNs G_1 and G_2, we let $G_1 \underset{L_T,L_P}{||} G_2$ denote the composition of G_1 and G_2 where the two Petri nets share those transitions in L_T and places in L_P. Refer to [1] for further details. For examples, we attach below those LGSPNs that correspond to the SimpleCopier components in Figure 3, where the white boxes represent timed transitions, and the black boxes are for immediate transitions. Notice that the LGSPNs in Figure 4 share some of their labels, specifically *putPage*$_{ev}$ and *getPage*$_{ev}$, with which the SimpleCopier LGSPN is defined as $(G_{Scanner} \underset{\emptyset,\emptyset}{||} G_{Printer}) \underset{\emptyset,L_P}{||} G_{Printer}$ where $L_P = \{putPage_{ev}, getPage_{ev}\}$.

3 Transformation of UML Design Models to Process Algebra

We define a transformation of UML to the stochastic process algebra IMC. The metamodel of the subset covers core parts of the *StateMachine* and the *Activity* metamodels. Since stochastic performance analysis based on UML *StateMachine* has already been discussed elsewhere [4], we focus the discussion on transformation of UML AD for performance analysis.

3.1 Transformation of Components to IMC

Each class in a UML model is defined by its behavior and operations, as depicted in Figure 1. For the convenience of our discussion, we introduce the following syntax that is equivalent with its graphical counterpart.

$$\begin{array}{rll} Class & ::= (Behavior, Operation, Operation, \ldots) & \textit{classifierBehavior} \text{ and operations} \\ Operation & ::= OperationName = Behavior & \text{Operation without parameters} \\ Behavior & ::= Activity \ \mid \ StateMachine & \end{array}$$

Based on this, we then define a new transformation $[\![\cdot]\!]$ from UML to the stochastic process algebra IMC.

$$[\![(B, f_1 = B_1, \ldots, f_k = B_k)]\!] = [\![B]\!] \ |_{\{f_1,\ldots,f_k\}} \ (P_1 \ |_\emptyset \ \cdots \ |_\emptyset \ P_k)$$

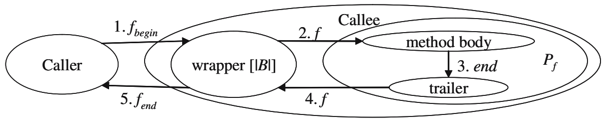
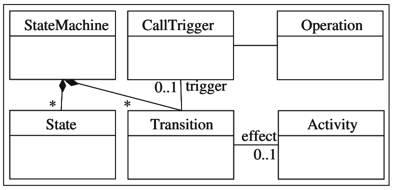


Fig. 5. IMC encoding of an operation invocation (f)



$SM ::= T, T, \ldots$	A *StateMachine* is a set of transitions
$T ::= S \overset{Tr/E}{\rightarrow} S$	Transition with trigger and effect
$S ::= \mathit{initial} \mid \mathit{final} \mid s$	State
$Tr ::= \emptyset \mid \mathit{trigger}(f)$	(possibly empty) *CallTrigger*
$E ::= \emptyset \mid A$	(possibly empty) *Effect* as an *Activity* where A is defined in Figure 7.

Fig. 6. Our UML *StateMachine* Metamodel (left) and Syntactic Notion (partial, right)

$$\begin{aligned}
\text{where}\quad P_i &\triangleq \mu X_i.(\widehat{f_i}.[\![B_i]\!] \mid_{\{end\}} (\widehat{end}.f_i.X_i)) \\
[\![B_i]\!] &\triangleq \cdots .end.\emptyset && \text{// the } i\text{-th operation} \\
[\![B]\!] &\triangleq \cdots .\widehat{f1_{begin}}.f1.\widehat{f1}.f1_{end}.\cdots + \cdots .\widehat{fi_{begin}}.fi.\widehat{fi}.fi_{end}.\cdots && \text{// wrapper of the operations}
\end{aligned}$$

The IMC representation of each operation $[\![B_i]\!]$ is prefixed with $\widehat{f_i}$ and synchronizes with the process $\widehat{end}.f_i.X_i$ by using the *end* action. This whole composition is further enclosed by a recursive operator μX_i. Therefore, on reaching the end of the execution of $[\![B_i]\!]$, f_i is issued to send notice of the completion to $[\![B]\!]$. Then, the composite process $\widehat{f_k}.[\![B_k]\!] \mid_{\{end\}} (\widehat{end}.f_k.X_k)$ is replicated so that it can accept the next invocation. When another component invokes f_i, the caller transmits $f_{i_{begin}}$, which is received by $[\![B]\!]$. Then, $[\![B]\!]$ dispatches the invocation to P_i. Conversely, $[\![B]\!]$ transmits $f_{i_{end}}$ to the caller after the reception of the completion notice from P_i (see Figure 5.) For example, Buffer is transformed as follows (transformations of activities and state machines will be defined in the following sections):

$$\begin{aligned}
\mathit{Buffer} &= (SM_{\mathit{Buffer}}, \mathit{putPage} = A_{\mathit{putPage}}, \mathit{getPage} = A_{\mathit{getPage}}) \\
&\quad \text{where } SM_{\mathit{Buffer}}, A_{\mathit{putPage}} \text{ and } A_{\mathit{getPage}} \text{ are defined as activity or state diagrams in Figure 3} \\
[\![\mathit{Buffer}]\!] &= P_{SM_{\mathit{Buffer}}} \mid_{\{\mathit{putPage}, \mathit{getPage}\}} (P_{\mathit{putPage}} \mid_{\emptyset} P_{\mathit{getPage}}) \text{ where} \\
P_{SM_{\mathit{Buffer}}} &\triangleq \mu X_{s_0}.\widehat{\mathit{putPage}}_{begin}.\mathit{putPage}.\widehat{\mathit{putPage}}.\mathit{putPage}_{end}.(\mu X_{s_1}.(\widehat{\mathit{getPage}}_{begin}.\mathit{getPage}.\widehat{\mathit{getPage}}.\mathit{getPage}_{end}.X_{s_0} + \\
&\quad \widehat{\mathit{putPage}}_{begin}.\mathit{putPage}.\widehat{\mathit{putPage}}.\mathit{putPage}_{end}.(\mu X_{s_2}.(\widehat{\mathit{getPage}}_{begin}.\mathit{getPage}.\widehat{\mathit{getPage}}.\mathit{getPage}_{end}.X_{s_1} + \\
&\quad \widehat{\mathit{putPage}}_{begin}.\mathit{putPage}.\widehat{\mathit{putPage}}.\mathit{putPage}_{end}.\widehat{\mathit{getPage}}_{begin}.\mathit{getPage}.\widehat{\mathit{getPage}}.\mathit{getPage}_{end}.X_{s_2})))) \\
P_{\mathit{putPage}} &\triangleq \mu X_{\mathit{putPage}}.(\widehat{\mathit{putPage}}.(\theta).end.\emptyset \mid_{\{end\}} \mathit{putPage}.X_{\mathit{putPage}}) \\
P_{\mathit{getPage}} &\triangleq \mu X_{\mathit{getPage}}.(\widehat{\mathit{getPage}}.(\theta).end.\emptyset \mid_{\{end\}} \mathit{getPage}.X_{\mathit{getPage}})
\end{aligned}$$

3.2 Transformation of *StateMachine* to IMC: A Brief Overview

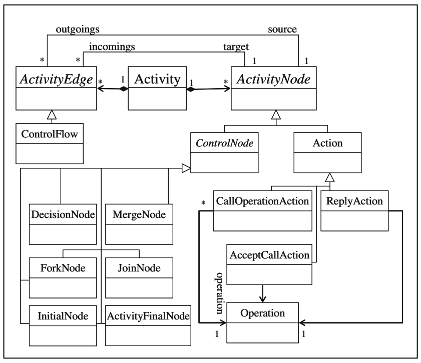
We consider a subset of *StateMachine* metamodel that consists of *States* and *Transitions*, where *States* include *initial* and *final* states, and a *Transition* may have at most one *Trigger* and/or *Effect*. For brevity, we consider only *CallTrigger* as a trigger. The semi-formal definition of *StateMachine* is shown below: Since existing work (e.g. [4])

already covers transformation from state machines to process algebra, we focus on the transformations of triggers and effects that are most relevant to our discussion. A call trigger *trigger*(f) waits for the operation f to be called, invoke f, waits for f to complete execution, and notifies the caller of its completion. Thus, the semantics of the trigger is defined as $\llbracket trigger(f) \rrbracket = \widehat{f_{begin}}.f.\widehat{f}.f_{end}.\emptyset$. If a transition has an effect, it is executed after the completion of the triggered operation. Thus the transformation becomes $\llbracket trigger(f)/A_e \rrbracket = \widehat{f_{begin}}.f.\widehat{f}.f_{end}.\llbracket A_e \rrbracket$. Refer to [19] for the detail.

3.3 A Subset of the UML Activity Diagrams

To highlight the key concept of the transformation, we chose a subset of the UML 2.0 *Activity* metamodel for our input language, which is shown in Figure 7. The constructs in the subset include *Action*, *ControlNode*, and *ControlFlow*. For *ControlNode*, *InitialNode* and *ActivityFinalNode* respectively indicate the initial and the final points of activities. *DecisionNode* (*MergeNode*) split (merge) control flows respectively, and *ForkNode* / *JoinNode* are employed for concurrency controls. We believe these provide sufficient support for designing components of concurrent systems in the early development phases.

For brevity, we further introduce the following constraints into activity diagrams: (i) each diagram must have exactly one *InitialNode* and at most one *ActivityFinalNode*, (ii) only *DecisionNode* and *ForkNode* can have multiple outgoing edges, (iii) only *MergeNode* and *JoinNode* can have multiple incoming edges, (iv) all the other nodes must have exactly one incoming edge and one outgoing edge, and (v) no edge can be associated with conditional information (a *guard*.) Notice that these constraints are not restrictive in terms of expressiveness. For example, the UML 2.0 specification defines that multiple outgoing flows from an action must be treated in the same way as those from a *ForkNode*. Therefore, such flows can be eliminated by introducing an additional *ForkNode* in a pre-processing step. In the rest of the discussion, we assume (a) each node of a diagram has a unique label and (b) each action node is annotated with an UML-SPT stereotype.



$N ::=$		labeled node
	$init_l$	*InitialNode*
	$\mid\ action(a, r)_l$	*Action* with name a and rate r
	$\mid\ call(f)_l$	*CallOperationAction* calling f
	$\mid\ accept(f)_l$	*AcceptCallAction* starting f
	$\mid\ reply(f)_l$	*ReplyAction* returning from f
	$\mid\ merge_l$	*MergeNode*
	$\mid\ join_l$	*JoinNode*
$A, B ::=$		labeled activity
	$Final_l$	*ActivityFinalNode*
	$\mid\ FwdMerge_l$	*MergeNode* (forward)
	$\mid\ BwdMerge_l$	*MergeNode* (backward)
	$\mid\ Join_l$	*JoinNode*
	$\mid\ N \rightarrow A$	*ControlFlow*
	$\mid\ Decision(A, B)_l$	*DecisionNode* with no guard
	$\mid\ Fork(A, B)_l$	*ForkNode*

Fig. 7. Our UML AD metamodel (left) and the Syntax of Activity Terms (right)

3.4 Activity Terms

We introduce an intermediate representation of our subset of UML AD, which we call *activity terms*. The formal syntax of activity terms is defined in Figure 7.

Most noticeably, the building blocks of activity terms are divided into two syntactic categories, namely *labeled nodes* and *labeled activities*. Schematically, *Actions* in activity diagrams are translated to nodes, whereas *ControlNodes* or *ControlFlows* are translated into activities. As exceptions to this, however, *MergeNodes* and *JoinNodes* have their counterparts both in the node and the activity categories. This duality comes from the fact that we derive activity terms from activity diagrams by means of depth-first graph traversals. Each time a new *MergeNode* (or a *JoinNode*) in an activity diagram that has never been visited is encountered during traversal, we derive from that node a merge_l (join_l) where l is a new label. Upon encountering a *MergeNode* (*JoinNode*) that has already been visited and marked with l, we then derive a $\mathit{FwdMerge}_l$ or $\mathit{BwdMerge}_l$ (Join_l) and terminates the current traversal path. The difference between *FwdMerges* and *BwdMerges* is that we derive a $\mathit{BwdMerge}_l$ if the encountered node was already included in the current traversal path, or otherwise derive a $\mathit{FwdMerge}_l$.

For example, let us consider the activity terms for SimpleCopier in Figure 3; The activity terms for Scanner and Printer are derived in the same way. Actually, the initial node in either diagram directly connects to a *MergeNode*. When visiting it from the initial node, we derive a *merge* with a fresh new label and travers to the following node. Later in the traversal, when the *MergeNode* is visited again, a *BwdMerge* is derived and the entire derivation terminates. Refer to the following equations for the detail.

$$\begin{aligned}\mathit{Scanner} &\triangleq \mathit{init}_{s1} \rightarrow \mathit{merge}_{s2} \rightarrow (\mathit{scan}, \lambda)_{s3} \rightarrow \mathit{call}(\mathit{put})_{s4} \rightarrow \mathit{BwdMerge}_{s2}\\ \mathit{Printer} &\triangleq \mathit{init}_{p1} \rightarrow \mathit{merge}_{p2} \rightarrow \mathit{call}(\mathit{get})_{p3} \rightarrow (\mathit{print}, \lambda')_{p4} \rightarrow \mathit{BwdMerge}_{p2}\end{aligned}$$

3.5 Transformation of Activity Term to IMC

We here define a transformation of activity terms to IMC processes. The transformation, $[\![\cdot]\!] : \mathit{Activity} \rightarrow \mathit{IMC}$, is semi-formally defined in Table 1.

Each activity term with a rate-annotated action prefix, $\mathit{action}(a, r)_l \rightarrow A$, is mapped to a IMC process $(r).[\![A]\!]$ after discarding the action name. For each call node, $\mathit{call}(f)_l$, two distinct IMC actions, f_{begin} and f_{end}, are generated for synchronous communications between the caller and the callee processes at the beginning and the end of the call operation, respectively. Note that for clarity we distinguish between the sender side (f_{begin}) and the receiver side ($\widehat{f_{\mathit{begin}}}$) of the communication, although the IMC semantics does not depend on such distinctions.

As discussed earlier, each activity term may simultaneously include a single merge_l node and an arbitrary number of $\mathit{FwdMerge}_l$ or $\mathit{BwdMerge}_l$ activities, all of which derive from the same *MergeNode*. When mapping $\mathit{merge}_l \rightarrow A$, a new process variable X_l is created to point to the resulting process ($X_l = [\![\mathit{merge}_l \rightarrow A]\!]$). In addition, the associations between l and $X_l/[\![\mathit{merge}_l \rightarrow A]\!]$ are memorized, which are referred when $\mathit{FwdMerge}_l$ or $\mathit{BwdMerge}_l$ are encountered later in the transformation. For example, by applying $[\![\cdot]\!]$ to the activity terms of *Scanner* and *Printer*, the following IMC processes for *Scanner* and *Printer* are obtained, which are slightly different from the previous

Table 1. Transformation of Activity Terms to IMC

$$\begin{aligned}
[\![\mathit{init}_l \rightarrow A]\!] &= [\![A]\!] \\
[\![\mathit{Final}_l]\!] &= \mathit{end}.\emptyset \\
[\![\mathit{action}(a,r)_l \rightarrow A]\!] &= (r).[\![A]\!] \\
[\![\mathit{call}(f)_l \rightarrow A]\!] &= f_{\mathit{begin}}.\widehat{f_{\mathit{end}}}.[\![A]\!] \\
[\![\mathit{accept}(f)_l \rightarrow A]\!] &= \widehat{f_{\mathit{begin}}}.[\![A]\!] \\
[\![\mathit{reply}(f)_l \rightarrow A]\!] &= f_{\mathit{end}}.[\![A]\!] \\
[\![\mathit{merge}_l \rightarrow A]\!] &= \mu X_l.[\![A]\!] \quad \text{where } X_l \text{ is a fresh variable labeled } l \\
& \ (\text{memorizing } \Pi(l) \triangleq \mu X_l.[\![A]\!] \text{ and } \chi(l) \triangleq X_l) \\
[\![\mathit{FwdMerge}_l]\!] &= \Pi(l) \\
[\![\mathit{BwdMerge}_l]\!] &= \chi(l) \\
[\![\mathit{join}_l \rightarrow A]\!] &= \begin{cases} \tau.[\![A]\!] & (\mathit{indegree}(l) = 1) \\ \underbrace{\widehat{\mathit{join}_l}.\cdots.\widehat{\mathit{join}_l}}_{\mathit{indegree}(l)-1}.[\![A]\!] & (\mathit{indegree}(l) > 1) \end{cases} \\
[\![\mathit{Join}_l]\!] &= \mathit{join}_l.\emptyset \\
[\![\mathit{Decision}(A_1,A_2)_l]\!] &= [\![A_1]\!] + [\![A_2]\!] \\
[\![\mathit{Fork}(A_1,A_2)_l]\!] &= \tau.([\![A_1]\!] \mid_L [\![A_2]\!]) \quad \text{where } L = \{\alpha \mid \alpha \in \mathit{act}([\![A_1]\!]) \wedge \widehat{\alpha} \in \mathit{act}([\![A_2]\!])\}
\end{aligned}$$

definitions in Section 2.2.

$$\begin{aligned}
[\![\mathit{Scanner}]\!] &= \mu X_{S2}.\left((\lambda).\mathit{putPage}_{\mathit{begin}}.\widehat{\mathit{putPage}_{\mathit{end}}}.X_{S2}\right) \\
[\![\mathit{Printer}]\!] &= \mu X_{P2}.\left(\mathit{getPage}_{\mathit{begin}}.\widehat{\mathit{getPage}_{\mathit{end}}}.(\lambda').X_{P2}\right)
\end{aligned}$$

For join-synchronization between n concurrent threads launched by preceding *Fork* activities, each of the $(n-1)$ terminating threads transmits join_l when it reaches the synchronization point, and the only remaining thread waits for them before resuming the rest of its computation. Note that auxiliary functions, *indegree* and *act*, are employed without definitions: $\mathit{indegree}(l)$ returns, for each node labeled l, the number of incoming edges for the node, and $\mathit{act}(P)$ accumulates the action names that appear in P.

3.6 Composition of Components

When an activity includes a $\mathit{call}(f)$ node, it is supposed to communicate with another component, whose behavior is defined either in an activity that includes $\mathit{accept}(f)$ / $\mathit{reply}(f)$ nodes, or a state machine that includes call triggers for f. Thus, in reality we need to transform a system of communicating components to a composite IMC process. In order to extend $[\![\cdot]\!]$ to transform a system of composed components, we first let $C_1 \bowtie_L C_2$ denote a system consisting of C_1 and C_2, where L is a set of call operation names. Formally, we introduce a new syntactic entity, S as follows:

$$S ::= C \mid S \underset{L}{\bowtie} S$$

For example, the SimpleCopier system is defined as a composite of classes as follows:

$$\mathit{SimpleCopier} \triangleq \mathit{Buffer} \underset{\{\mathit{putPage},\mathit{getPage}\}}{\bowtie} \left(\mathit{Scanner} \underset{\emptyset}{\bowtie} \mathit{Printer}\right)$$

Such a composition can be derived in several ways, for example by using communication diagrams [4]. Alternatively, the target attribute of a *CallOperationAction* can be used to resolve caller-callee relationships, if the values of attributes can be statically determined. Once a system of components is defined using $\bowtie$, its transformation is defined in a straightforward manner as follows:

$$[\![S_1 \underset{\{f_1,\ldots,f_k\}}{\bowtie} S_2]\!] = [\![S_1]\!] \mid_L [\![S_2]\!] \text{ where } L = \{f_{begin}, f_{end} \mid f = f_1, \ldots, f_k\}$$

4 Equivalence Between the LGSPN Semantics and Our Semantics

In this section, we show the validity of our process algebraic semantics for UML AD in the following steps. First, we define the LGSPN semantics of IMC (Section 4.1), along with the LGSPN semantics of the activity terms (Section 4.2). Then we show the equivalence between our IMC transformation $[\![\cdot]\!]$ and the LGSPN transformation $(\!|\cdot|\!)$ in the sense that the following diagram commutes. That is to say, given an activity term A, $\psi([\![A]\!]) = (\!|A|\!)$ holds without any conditions.

$$\begin{tikzcd}
\text{Activity Term: } A \arrow[r, "{[\![\ ]\!]}"] \arrow[dr, "{(\!|\ |\!)\ (\S4.2)}"'] \arrow[dr, phantom, shift left=3ex, "\circlearrowright"] & \text{IMC: } [\![A]\!] \arrow[d, "\psi\ (\S4.1)"] \\
 & \text{LGSPN: } (\!|A|\!) = \psi([\![A]\!])\ (\S4.3)
\end{tikzcd}$$

Since this section discusses only UML AD, we restrict our language of composed components to the following form:

$$S ::= A \mid S \bowtie S$$

4.1 LGSPN Semantics of IMC

The stochastic process algebras have strong relationships with the stochastic Petri net formalism. They indeed share the same underlying mathematical foundation of Markovian theories. Based on this commonality, Ribaudo [15] has defined a mapping from PEPA to LGSPN. Owing much to this, we define a mapping ψ from each IMC process P to an LGSPN $\psi(P) = (P, T, F, R, M_0, L)$, which is summarized in Table 2.

In the definition of ψ, we have introduced two special labels, *init*$_l$ and *final*, which implicitly designate the initial and the final nodes of the Petri-net graphs. Each process with a prefix action, $(r).P$ or $\alpha.P$, is encoded to a composition (superposition) of two LGSPNs which correspond to the prefix part and the remaining part of the process, respectively. Note that the initial marking M_0 of $\psi(P)$ is redefined so that the all places in M_0 are regarded as identical with p_2, the terminating place of G, and thus folded to a single place. For composite processes, $\psi(P_1 + P_2)$ is defined as a composition of $\psi(P_1)$ and $\psi(P_2)$, of which the initial nodes are folded into a single place that is labeled l. $\psi(P_1 \mid_L P_2)$ is also defined as a composition of $\psi(P_1)$ and $\psi(P_2)$, though in this case the transitions specified by L are superposed while the places are simply merged together without superposition. See [15] for the details.

Table 2. LGSPN Semantics of IMC (almost identical with those in [15])

$$
\begin{aligned}
\psi(\emptyset) &= (\{p\}, \emptyset, \emptyset, \emptyset, \{p\}, \{p \mapsto \mathit{final}\}) \text{ where } p \text{ is a fresh place} \\
\psi((r).P) &= G \mathop{||}_{\emptyset,\{l\}} \psi(P)[\{M_0 \mapsto l\}] \\
\text{where} \quad & G = (\{p_1, p_2\}, \{t\}, \{(p_1, t), (t, p_2)\}, \{t \mapsto r\}, \{p_1\}, \{t \mapsto \tau, p_2 \mapsto l\}) \\
& M_0 \text{ is the initial marking of } \psi(P), l \text{ is a fresh label} \\
\psi(\alpha.P) &= G \mathop{||}_{\emptyset,\{l\}} \psi(P)[\{M_0 \mapsto l\}] \\
\text{where} \quad & G = (\{p_1, p_2\}, \{t\}, \{(p_1, t), (t, p_2)\}, \{t \mapsto \infty\}, \{p_1\}, \{p_1 \mapsto \mathit{init}_l, t \mapsto \tilde{\alpha}, p_2 \mapsto l\}) \\
& M_0 \text{ is the initial marking of } \psi(P), l \text{ is a fresh label} \\
\psi(P_1 + P_2) &= \psi(P_1)[\{M_{01} \mapsto l\}] \mathop{||}_{\mathit{Lab}(\psi(P_1)) \cup \mathit{Lab}(\psi(P_2)), \{l\}} \psi(P_2)[\{M_{02} \mapsto l\}] \\
\text{where} \quad & M_{01}, M_{02} \text{ are the initial markings of } \psi(P_1) \text{and} \psi(P_2), l \text{ is a fresh label} \\
\psi(X_l) &= (\{p_{X_l}\}, \emptyset, \emptyset, \emptyset, \{p_{X_l}\}, \{p_{X_l} \mapsto \mathit{init}_l\}) \text{ where } p_{X_l} \text{ is a fresh place} \\
\psi(\mu X_l.P) &= (P \setminus P_l, T, F \setminus \{(t, p) | p \in P_l\} \cup \{(t, M_0) | p \in P_l \wedge (t, p) \in F\}, R, M_0, L) \\
& \text{where } \psi(P) = (P, T, F, R, M_0, L), P_l = \{p | p \in P \wedge L(p) = \mathit{init}_l\} \\
\psi(P|_L Q) &= (P_1 \cup P_2, (T_1 \setminus T_1^L) \cup (T_2 \setminus T_2^L) \cup T_{1\times 2}^L, \\
& (F_1 \setminus F_1^L) \cup (F_2 \setminus F_2^L) \cup F_{1\times 2}^L, R_1 \cup R_2, M_{01} \cup M_{02}, L) \\
\text{where} \quad & \psi(P) = (P_1, T_1, F_1, R_1, M_{01}, L_1), \psi(Q) = (P_2, T_2, F_2, R_2, M_{02}, L_2) \\
& T_1^L = \{t \in T_1 | L_1(t) \in L\}, T_2^L = \{t \in T_2 | L_2(t) \in L\} \\
& T_{1\times 2}^L = \{t_i \times t_j | t_i \in T_1^L, t_j \in t_2^L, L_1(t_i) = L_2(t_j)\} \\
& F_1^L = \{(p, t), (t, p) \in F_1 | L_1(t) \in L\}, F_2^L = \{(p, t), (t, p) \in F_2 | L_2(t) \in L\} \\
& F_{1\times 2}^L = \{(p, t_i \times t_j) | t_i \times t_j \in T_{1\times 2}^L \wedge ((p, t_i) \in F_1 \vee (p, t_j) \in F_2)\} \\
& \quad \cup \{(t_i \times t_j, p) | t_i \times t_j \in T_{1\times 2}^L \wedge ((t_i, p) \in F_1 \vee (t_j, p) \in F_2)\} \\
& L(t) = L_1(t) \text{ (if } t \in T_1 \setminus T_1^L), L_2(t) \text{ (if } t \in T_2 \setminus T_2^L), \tau \text{ (if } t \in T_{1\times 2}^L)
\end{aligned}
$$

4.2 LGSPN Semantics of Activity Terms

As mentioned earlier, many attempts have been made to establish stochastic Petri net semantics for UML diagrams with performance-related annotations. Among those attempts, [12] introduced a set of rules to transform a UML AD to an LGSPN. Thanks to the labeling functionality of LGSPN, this successfully maps each set of activities, connected with each other according to their caller-callee relations, to a single (superposed) Petri net. With slight modifications, we define transformation rules for our activity terms, as shown in Figure 8 and Table 3.

The rules in Figure 8 define mapping of the nodes of activity terms, defined in Section 3.2, to an LGSPN, most of which are identical with those that appear in [12]. Note that, since UML 1.5 (used in [12]) does not support *AcceptCallAction* nor *ReplyAction*, we instead adopted the rules for *Signal Receipt* and *Signal Sending* in [12] for these two actions. Exploiting these, the rules in Table 3 define a mapping of activity terms to an LGSPN.

4.3 Equivalence Between $[\![\cdot]\!]$ and $(\!|\cdot|\!)$

We are now ready to show the equivalence between our stochastic process algebraic semantics $[\![\cdot]\!]$ and the stochastic Petri net semantics $(\!|\cdot|\!)$ by proving $\psi \circ [\![\cdot]\!] = (\!|\cdot|\!)$. To this end, we first define the equality between LGSPNs in terms of the underlying mathematical foundation:

Definition 2 (Equality and Equivalence between LGSPNs). *Two LGSPNs G_1 and G_2 are* equal *($G_1 = G_2$) when their underlying continuous-time Markov chains*

(CTMCs) are identical with each other. Meanwhile, G_1 and G_2 are equivalent ($G_1 \equiv G_2$) when their graph structures and rate assignments are identical.

Action: $\llparenthesis action(a,r)_l \rrparenthesis$		*CallOperationAction*: $\llparenthesis call(f)_l \rrparenthesis$	
$p1\|ini_l$ $t1\|out_r$ $p2\|execute$ $t2\|cond_{ev}$ $p3\|ini_{head(A)}$	$p1\|ini_l$ $t1\|out_r$ $p2\|execute$ $t2\|cond_{ev}$ $p3\|waiting$ $t3\|join_{l'}$	$p1\|init_l$ $t1\|cond_{ev}$ $p2\|ev_f$ $p3\|waiting$ $p4\|ack_f$ $t2\|out$ $p5\|init_{head(A)}$	$p1\|init_l$ $t1\|cond_{ev}$ $p2\|ev_f$ $p3\|waiting$ $p4\|ack_f$ $t2\|out$ $p5\|waiting$ $t3\|join_{l'}$
(1-a.) next node is not a *join*	(1-b.) next node is $join_{l'}$	(2-a.) next node is not a *join*	(2-b.) next node is $join_{l'}$
MergeNode: $\llparenthesis merge_l \rrparenthesis$		*JoinNode*: $\llparenthesis join_l \rrparenthesis$	*ActivityFinalNode:* $\llparenthesis Final_l \rrparenthesis$
$p1\|init_l$ $t1\|do\_merge$ $p2\|init_{head(A)}$	$p1\|init_l$ $t1\|join_{l'}$	$t1\|join_l$ $p1\|init_{head(A)}$	$p1\|init_l$ $t1\|ending$ $p2\|end_A$
(3-a.) next node is not a *join*	(3-b.) next node is $join_{l'}$	(4.) *JoinNode*	(5.) final node
AcceptCallAction: $\llparenthesis accept(f)_l \rrparenthesis$		*ReplyAction*: $\llparenthesis reply(f)_l \rrparenthesis$	
$p2\|ev_f$ $p1\|init_l$ $t1\|acc_f$ $p3\|init_{head(A)}$	$p2\|ev_f$ $p1\|init_l$ $t1\|acc_f$ $p3\|waiting$ $t3\|join_{l'}$	$p2\|ack_f$ $p1\|init_l$ $t1\|ret_f$ $p3\|init_{head(A)}$	$p2\|ret_f$ $p1\|init_l$ $t1\|acc_f$ $p3\|waiting$ $t3\|join_{l'}$
(6-a.) next node is not a *join*	(6-b.) next node is $join_{l'}$	(7-a.) next node is not a *join*	(7-b.) next node is $join_{l'}$

Fig. 8. Transformation $\llparenthesis\cdot\rrparenthesis$ of Activity Term Nodes N to an LGSPN [12]

Table 3. Transformation $\llparenthesis\cdot\rrparenthesis$ of Activity Term A to LGSPN [12]

$$
\begin{array}{rcl}
\llparenthesis init_l \rightarrow A \rrparenthesis & = & \llparenthesis A \rrparenthesis \\
\llparenthesis Final_l \rrparenthesis & = & \text{as in Figure 8 (5.)} \\
\llparenthesis action(a,r)_l \rightarrow A \rrparenthesis & = & \llparenthesis action(a,r)_l \rrparenthesis \mathop{||}_{\{join_{head(A)}\},\{init_{head(A)}\}} \llparenthesis A \rrparenthesis \\
\llparenthesis call(f)_l \rightarrow A \rrparenthesis & = & \llparenthesis call(f)_l \rrparenthesis \mathop{||}_{\{join_{head(A)}\},\{init_{head(A)}\}} \llparenthesis A \rrparenthesis \\
\llparenthesis accept(f)_l \rightarrow A \rrparenthesis & = & \llparenthesis accept(f)_l \rrparenthesis \mathop{||}_{\{join_{head(A)}\},\{init_{head(A)}\}} \llparenthesis A \rrparenthesis \\
\llparenthesis reply(f)_l \rightarrow A \rrparenthesis & = & \llparenthesis reply(f)_l \rrparenthesis \mathop{||}_{\{join_{head(A)}\},\{init_{head(A)}\}} \llparenthesis A \rrparenthesis \\
\llparenthesis merge_l \rightarrow A \rrparenthesis & = & \llparenthesis merge_l \rrparenthesis \mathop{||}_{\{join_{head(A)}\},\{init_l, init_{head(A)}\}} \llparenthesis A \rrparenthesis \\
 & & (\text{memorizing } pn(l) \triangleq \llparenthesis merge_l \rightarrow A \rrparenthesis) \\
\llparenthesis FwdMerge_l \rrparenthesis & = & pn(l) \\
\llparenthesis BwdMerge_l \rrparenthesis & = & (\{p\}, \emptyset, \emptyset, \emptyset, \emptyset, \{p \mapsto init_l\}) \\
\llparenthesis join_l \rightarrow A \rrparenthesis & = & \llparenthesis join_l \rrparenthesis \mathop{||}_{\emptyset,\{init_{head(A)}\}} \llparenthesis A \rrparenthesis \\
\llparenthesis Join_l \rrparenthesis & = & (\emptyset, \{t\}, \emptyset, \{t \mapsto \infty\}, \emptyset, \{t \mapsto join_l\}) \\
\llparenthesis Decision(A_1, A_2)_l \rrparenthesis & = & (\llparenthesis A_1 \rrparenthesis [\{M_{01} \mapsto init_l\}] \mathop{||}_{LT,\{init_l\}} \llparenthesis A_2 \rrparenthesis)[\{M_{02} \mapsto init_l\}] \\
\text{where} & & LT = \{join_{l'} | l' \in Lab(Decision(A_1, A_2)_l)\},\ M_{01}, M_{02} \text{ are initial markings of } \llparenthesis A_1 \rrparenthesis, \llparenthesis A_2 \rrparenthesis \\
\llparenthesis Fork(A_1, A_2)_l \rrparenthesis & = & G \mathop{||}_{\emptyset,\{init_{head(A_i)}\}} (\llparenthesis A_1 \rrparenthesis \mathop{||}_{LT,LP} \llparenthesis A_2 \rrparenthesis) \\
\text{where} & & G = (\{p, p_1, p_2\}, \{t\}, \{(p,t), (t,p_1), (t,p_2)\}, \{t \mapsto \infty\}, \{p\}, \\
 & & \quad \{p \mapsto init_l, p_1 \mapsto init_{head(A_1)}, p_2 \mapsto init_{head(A_2)}\}) \\
 & & LT = \{join_{l'} | l' \in Lab(Fork(A_1, A_2)_l)\},\ LP = \{init_{l'} | l' \in Lab(Fork(A_1, A_2)_l)\} \\
\llparenthesis S_1 \mathop{\bowtie}_{f_1,\ldots,f_k} S_2 \rrparenthesis & = & \llparenthesis S_1 \rrparenthesis \mathop{||}_{\emptyset,\{ev_f, ack_f | f = f_1,\ldots,f_k\}} \llparenthesis S_2 \rrparenthesis
\end{array}
$$

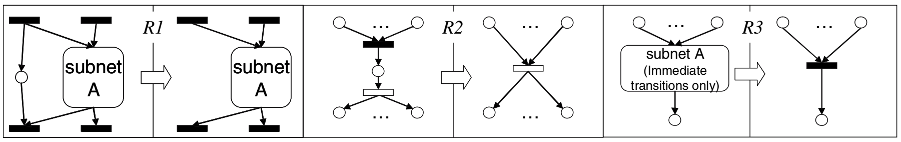


Fig. 9. Rules $(R1)$, $(R2)$, and $(R3)$ [17]

Notice that $G_1 \equiv G_2$ obviously implies $G_1 = G_2$. Unfortunately, we cannot expect $\psi \circ [\![\cdot]\!] \equiv (\!|\cdot|\!)$, since $\psi \circ [\![\cdot]\!]$ and $(\!|\cdot|\!)$ often generate structurally different LGSPNs. This leads us to graph-rewriting techniques, which help reduce each LGSPN G to a simpler form G' while guaranteeing their equality ($G = G'$). For the discussion in this section, we need the three rules depicted in Figure 9, which have been proven in [17] to meet the equality criteria: Given a LGSPN G, (*R1*) removes a place that does not affect the stochastic behavior of G, (*R2*) removes an immediate transition for which the succeeding transition is timed, and (*R3*) aggregates immediate transitions and replaces them with a single immediate transition. Refer to [17] for the details.

In Section 3.4, we have defined the composition of activities, which establishes caller-callee relationships among them. As a consequence, we can safely restrict UML AD, without sacrificing expressiveness, to those composite activities in which each *CallOperationAction* always has its corresponding *AcceptCallAction* and *ReplyAction*. According to [10], composite activities are called *complete* when they meet this restriction. In terms of activity terms, we formally define this as follows.

Definition 3. *A composite activity term S is* complete *iff LTS$([\![S]\!])$ has no action from $\widehat{\Sigma}$, where LTS(P) denotes a flat labeled transition system obtained by expansion of P.*

With the above definitions, we finally state the following proposition and its corollary. Since we are concerned only with complete composite activity terms, the corollary will directly support our claim that $\psi \circ [\![\cdot]\!] = (\!|\cdot|\!)$.

Proposition. Given a composite activity term S that is complete, $\psi([\![S]\!])$ and $(\!|S|\!)$ can be rewritten by rules in Figure 9 to LGSPNs that are equivalent to each other. In other words, there exist two sequences of rules $R_{i_1}, \ldots, R_{i_k}$ and $R_{j_1}, \ldots, R_{j_m}(1 \leq i_1, \ldots, i_k, j_1, \ldots, j_m \leq 3)$ s.t. $\psi([\![S]\!]) \overset{R_{i_1}}{\rightarrow} \cdots \overset{R_{i_k}}{\rightarrow} G$, $(\!|S|\!) \overset{R_{j_1}}{\rightarrow} \cdots \overset{R_{j_m}}{\rightarrow} G'$, and $G \equiv G'$.

Proof. The proof proceeds by induction on the structure of S. [***Final***] If $S \equiv$ *Final*, then $\psi([\![S]\!]) \equiv (\!|S|\!)$ holds without any rewriting. [***action***$(a, r) \rightarrow A$] The $(t2|cond_{ev})$ transition, generated by $(\!|$*action*$(a, r)|\!)$, has no corresponding transition in $\psi([\![S]\!])$. We consider the following two cases for this situation: If A is of the form *action*$(a', r') \rightarrow A'$, then (*R2*) eliminates $(t2|cond_{ev})$ so $\psi([\![S]\!]) \equiv (\!|S|\!)$. Otherwise, (*R4*) does the elimination. [***call***(f)/***accept***(f)/***reply***(f)] These contradict the assumption that S is complete (hence the claim holds). See also the $\bowtie$ case. [***merge*** $\rightarrow A$] The $(t|do\_merge)$ transition, generated by $(\!|$*merge* $\rightarrow A|\!)$, has no corresponding transition in $\psi([\![S]\!])$. We can exploit the same techniques as with [*action*$(a, r) \rightarrow A$]. [***FwdMerge***] same as with *merge* $\rightarrow A$. [***BwdMerge***] No rewriting is needed. [$\boldsymbol{join_l} \rightarrow A$] In case *indegree*$(l) = 1$, $\psi([\![S]\!]) \equiv (\!|S|\!)$ holds without any rewriting. Otherwise (*indegree*$(l) > 1$), $[\![S]\!]$ turns out to be incomplete, which contradicts the assumption. See also the *Fork* case. [***Join***$_l$] No rewriting is needed. [***Decision***$(\boldsymbol{A_1}, \boldsymbol{A_2})$] By the induction hypothesis, there exist $G_{1i} and G_{2i}$ $(i = 1, 2)$ s.t. $\psi([\![A_i]\!]) \overset{R1,R2,R3}{\longrightarrow} G_{1i}$, $(\!|A_i|\!) \overset{R1,R2,R3}{\longrightarrow} G_{2i}$, and $G_{1i} \equiv G_{2i}$. Since $\psi([\![S]\!])$ superposes G_{11} and G_{12} exactly in the same way as $(\!|S|\!)$ does G_{21} and G_{22}, we only need to show $G_{11} \underset{\emptyset, L_P}{||} G_{12} \equiv G_{21} \underset{\emptyset, L_P}{||} G_{22}$, which is the case. [***Fork***$(\boldsymbol{A_1}, \boldsymbol{A_2})$] We only need to consider the *join* nodes in S. For each $join_l$ that appears in S, *indegree*(l) becomes equal with $\|\{Join_l \mid Join_l \text{ appears in } S\}\| + 1$ and superpositions in $\psi([\![S]\!])$ and $(\!|S|\!)$ occur between the transitions deriving from $join_l$ and those from $Join_l$. When *indegree*$(l) = 2$, the claim holds since S includes only one $Join_l$ activity. Otherwise (*indegree*$(l) > 2$), so it turns out $(\!|S|\!)$ derives a single immediate transition from $join_l$ whereas $\psi([\![S]\!])$ derives a set of immediate transitions, which can be folded into one by (*R3*). [$\boldsymbol{S_1} \underset{\{f_1,\ldots,f_k\}}{\bowtie} \boldsymbol{S_2}$] The differences between $\psi([\![S]\!])$ and $(\!|S|\!)$ lie in the superpositions of those places/transitions that derive from $f_1, \ldots, f_k$. For each superposition, $(\!|S|\!)$ generates 3 extra places and 2 extra immediate transitions that do not correspond to any in $\psi([\![S]\!])$. Fortunately, the differences can be resolved by successive applications of (*R1*) and (*R3*). For further details, please refer to [19]. □

Corollary. Given a composite activity term S, $\psi([\![S]\!]) = (\!|S|\!)$ if S is complete.

5 Preliminary Evaluation

We implemented a prototype version of the proposed model transformation mechanism on top of the Eclipse platform [7], which provides a Java component of the UML 2.0 metamodel. Our implementation consists of five main components: A subset of the UML-SPT profile, a model transformer, a discrete event simulator (DES), a Markov-chain solver, and a simple graphical user interface. All of them are seamlessly integrated into the Eclipse platform. For each UML model, performance analysis starts with annotation of actions in the activity diagrams with performance-related properties in UML-SPT. These annotations may include symbolic variables, the values of which are defined separatedly in parameter DB files. Procedures for performance analysis can be defined as a program in scripting languages, including Perl and VBA, which specifies a UML model and a set of parameter DB files, and invokes transformation and execution of the model. Each script program also specifies how to present execution results. They can be shown within the Eclipse environment, or can be transferred to external tools like Excel.

We designed a hypothetical on-line shopping system, whose static structure is defined on the left of Figure 10 as a class diagram. The system includes one server (CPU) and three databases. OrderingSystem is an external sub-system for stock checking and shipping. Each function of these components was modeled as an activity diagram with a single timed action. In contrast, ServerSession is a software component that communicates directly with users. The operations of ServerSession are modeled as sequential

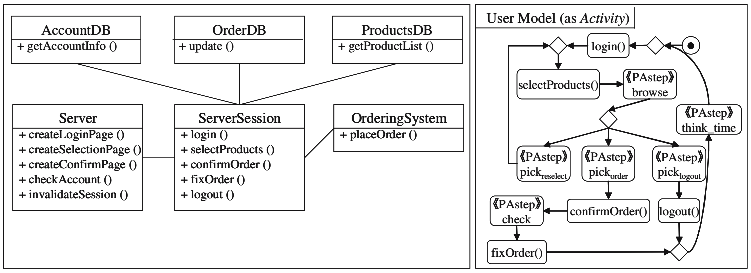


Fig. 10. Class Diagram of On-line Shopping System and a Usage Scenario

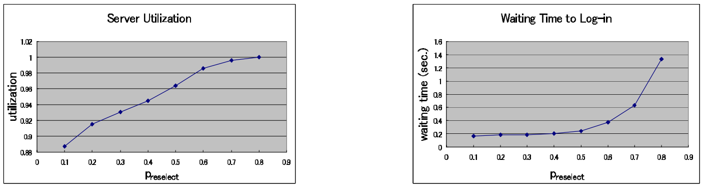


Fig. 11. Evaluation Results

accesses to the server and/or databases. A user begins shopping by sending a login request to the system. After logging in, the user asks for a list of available products and browses it for a while. Then the user chooses one of (i) cancel the purchase by logging-out with probability p_{logout}, (ii) request another product list and continue shopping with probability $p_{reselect}$, or (iii) check and confirm the order with $1-(p_{logout}+p_{reselect})$. This probabilistic choice is modeled as timed actions with small and ignorable delays. The user initiates another session $d_{thinktime}$ time units after the end of one session.

We evaluated utilization of the server and average waiting time for login requests of users with respect to the probability $p_{reselect}$. The target model contained 20 instances of user and ServerSession. The results are shown in Figure 11. As the value of $p_{reselect}$ increases, i.e. a user consumes more and more server resources, utilization of the server gets saturated and users are kept waiting longer and longer before they can log in.

6 Concluding Remarks

In this paper we have defined a stochastic process algebraic semantics for a subset of the UML metamodel, which primarily focuses on UML AD and the composition of components. The primary objective is to apply the MDD methodology for model-driven performance analysis, targeting in particular design verification in early development stages. The semantic function $[\![\cdot]\!]$ that relates UML models with UML-SPT annotations and stochastic process algebra IMC has successfully established a model transformation of UML for performance analysis. Further, we have proven that our semantics for UML AD is indeed equivalent to an existing Petri net semantics, which guarantees the validity of our approach.

There have been a large number of efforts to define formal semantics of UML: Störrle [18] formalized the Petri net semantics of a subset of UML 2.0 AD. He uses what he calls the "procedural Petri net semantics" as his semantic basis, which is not suitable for representing *shared resources* that we wanted to address. In [5], Canevet et al. demonstrated a performance analysis of UML 2.0 AD. by means of PEPA Nets [8]. The mapping to PEPA Nets was, however, defined in an intuitive fashion. Canevet et al. [4] analyzed UML 2.0 *StateMachine* by transforming them to PEPA. Their annotation language is, however, not compatible with UML-SPT or other related standards. Bolton et al. [2] proposed CSP semantics for UML 1.5 AD that primarily focused on functional aspects of tha models.

As possible future research directions, we are currently considering the following possibilities: One of the most important problems is how to handle other sorts of behavioral descriptions, in particular *Interactions*. They have already been considered by Bernardi et al. [1] in the UML 1.5 context. We are not yet quite sure to what degree these results can be applied to the UML 2.0 specification. Alternately, it would be both theoretically and practically interesting to extend the notion of duration from negative-exponential distributions to generic pdfs. Non-Markovian process algebras would be applicable for this purpose. Finally, feasibility studies with more realistic and complicated system models are necessary.

References

1. Simona Bernardi et al. From UML sequence diagrams and statecharts to analysable petri net models. In *Proc. 3rd Intl. Workshop on Software and Performance*, pages 35–45, 2002.
2. Christie Bolton and Jim Davies. Activity graphs and processes. In *Proc. IFM2000*, 2000.
3. Ed Brinksma and Holger Hermanns. Process algebra and markov chains. 2002.
4. C. Canevet et al. Performance modelling with UML and stochastic process algebras. In *Proc. UK Performance Engineering Workshop*, July 2002.
5. C. Canevet et al. Analysing UML 2.0 activity diagrams in the software performance engineering process. In *Proc. 4th Intl. Workshop on Soft. and Perf.*, pages 74–78, 2004.
6. Susanna Donatelli and Giuliana Franceschinis. The PSR methodology: Integrating hardware and software models. In *ICATPN'96*, pages 133–152. Springer-Verlag, June 1996.
7. Eclipse.org. Eclipse.org homepage. http://www.eclipse.org, 2005.
8. Stephen Gilmore et al. PEPA nets: A structured performance modelling formalism. In *Proc. TOOLS 2002*, April 2002.
9. Norbert Götz, Ulrich Herzog, and Michael Rettelbach. TIPP-a language for timed processes and performance evaluation. Technical report, University of Frlangen-Nurnherg, 1992.
10. Jane Hillston. *A compositional approach to performance modelling*. CUP, 1996.
11. D. Kartson, G. Balbo, S. Donatelli, G. Franceschinis, and Giuseppe Conte. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, Inc., 1994.
12. Juan Pablo López-Grao et al. From UML activity diagrams to stochastic petri nets: application to software performance engineering. *SIGSOFT Softw. Eng. Notes*, 29(1), 2004.
13. OMG. *UML Profile for Schedulability, Performance, and Time, v1.0*, September 2003.
14. OMG. *UML 2.0 Superstructure specification*, October 2004.
15. M. Ribaudo. Stochastic Petri net semantics for stochastic process algebras. In *Proc. PNPM95*, 1995.
16. Bran Selic. The Pragmatics of Model-Driven Development. *Software*, pages 19–25, 2003.

17. Carla Simone and Marco Ajmone Marsan. The application of EB-equivalence rules to the structural reduction of GSPN models. *J. Parallel Distrib. Comput.*, 15(3):296–302, 1992.
18. Harald Störrle. Semantics of control-flow in UML 2.0 activities. In *Proc. VL/HCC'04*, September 2004.
19. Naoshi Tabuchi, Naoto Sato, and Hiroaki Nakamura. Model-driven performance analysis of uml design models based on stochastic process algebra. Technical Report RT0604, 2005.
20. Jan Trowitzsch, Armin Zimmermann, and Günter Hommel. Towards quantitative analysis of real-time uml using stochastic petri nets. In *IPDPS*. IEEE Computer Society, 2005.

MDA Components: A Flexible Way for Implementing the MDA Approach*

Reda Bendraou[1], Philippe Desfray[2], and Marie-Pierre Gervais[1, 3]

[1] Laboratoire d'Informatique de Paris 6, 8 rue du Capitaine Scott - F75015 Paris
[2] Softeam
144 Av. des champs Elysées F75008 PARIS
`Philippe.Desfray@softeam.fr`
[3] Université Paris X
`{Reda.Bendraou, Marie-Pierre.Gervais}@lip6.fr`

Abstract. As the Model Driven Development (MDD) and Product Line Engineering (PLE) appear as major trends for reducing software development complexity and costs, an important missing stone becomes more visible: there is no standard and reusable assets for packaging the know-how and artifacts required when applied these approaches. To overcome this limit, we introduce in this paper the notion of MDA Component, i.e., a packaging unit for encapsulating business know-how and required resources in order to support specific operations on a certain kind of model. The aim of this work is to provide a standard way for representing this know-how packaging units. This is done by introducing a two-layer MOF-compliant metamodel. Whilst the first layer focuses on the definition of the structure and contents of the MDA Component, the second layer introduces a language independent way for describing its behavior. For a full specification, both layers can be merged using the UML2.0 package merge facility.

Keywords: MDA, MDD, PLE, Packaging of know-how, MDA Component, reusability.

1 Introduction

In software industry, growing expectations for reliable software in a short time to market makes development processes increasingly complex. The continuing evolution of technologies and the need for sophisticated information systems will not improve this situation. Then, it becomes more and more difficult for companies to respect deadlines and to provide software with the expected functionalities. According to the Standish Group CHAOS study report, in 2004, when 29% of projects succeeded (time, budget and required functions), among 53% are challenged (late, over budget and/or with less than the required features and functions); and 18% have failed (cancelled prior to completion or delivered and never used) [21]. Besides, if we look closer at the development processes, we can notice that it exists a convergence in the

* This work is supported in part by the IST European project "ModelWare" (contract no 511731).

practice of software production. Most of the time, developers and designers apply the same steps, handle similar tools and apply the identical tests. Unfortunately, this know-how is generally scattered and not capitalized. In order to face these lacks, two major approaches appear as a mean of reducing the software development complexity: on one hand, the MDD (Model Driven Development) approach by promoting the use of models as main actors of the software development cycle. On the other hand, Product Line Engineering (PLE) that promotes reusability through the specification of models covering a family of software products.

1.1 Model Driven Development

The MDD (Model Driven Development) is an approach to software development where extensive models are created before source code is written. By considering models as first class entities, MDD aims at reducing software production complexity. A primary example of MDD is the Object Management Group (OMG)'s Model Driven Architecture (MDA) initiative [13]. The MDA advocates the distinction between models designed independently of any technical considerations of the underlying platform i.e. PIM (Platform Independent Model) and models that include such considerations i.e. PSM (Platform Specific Model). In order to ensure a model driven approach for software development, a growing family of standards for representing variety of domain specific models emerges. Examples of such standards are UML (Unified Modeling language) [22][23], MOF (Meta Object Facility) [15], SPEM (Software Process Engineering Metamodel) [20], EDOC (Enterprise Distributed Object Computing) [7], etc. Besides, panoply of MDA-compliant tools providing developers with operation on models appears [12]. Nevertheless, the expert's know-how, domain specific features i.e. helpers, configuration parameters, libraries etc, and customized actions and knowledge applied on these models are provided neither by MDA standards nor by existing MDA-compliant tools. One can advocates that a solution would be to customize a tool in order to support all features needed for a specific context. Then, tool-vendors would have to adapt their tools every time a new profile, a new standard, operation on models or a new platform appears. This cannot be a long term solution.

1.2 Production Lines

The goal of Product Line Engineering (PLE) is to support the systematic development of a set of similar software systems. Main assets in PLE are the architecture model of the Product Line (PL) and the steps required i.e. the process model for defining it. Defining the PL architecture roughly consists in consolidating all the common features of a family of software products within a reusable core and packaging simple parts for customization [16]. Two of the fundamental needs in defining architecture for a PL are: 1) to be able to generalize or abstract from the individual products to capture the important aspects of the PL and 2) to be able to instantiate an individual product architecture from the PL architecture [19]. When MDA offers a set of standards for representing a reusable PL architecture model, it is far away of providing in a reusable format, the know-how, artifacts and resources needed for building and for instantiating this model.

In summary, both MDD and PLE lack of reusable and standard entities that can capitalize and encapsulate the knowledge and required resources when promoting these approaches, a key condition for a more cost-effective software production. To overcome this limit, we introduce in this paper the notion of MDA Component (MDAC), i.e., a packaging unit which encapsulates business know-how and required resources in order to support specific operations on a certain kind of models. One objective of MDACs is to provide industrialists with a standard and powerful way of representing their know-how. MDACs are autonomous, platform-independent and promote MDA standards. Design methodologies and modelling activities can be represented and exchanged in a standard format thanks to MDACs. Another objective of MDACs is that tools can be customized to a specific domain. They will be able to support new functionalities simply by integrating MDACs to their environment. Thus, tool-vendors will have to integrate one MDAC related to every new context rather than redesigning their tools in order to take this context into account.

MDA standards used conjunctly with MDACs can ensure PLE needs. Indeed, MDA offers a set of standards aiming at abstracting the domains that have to be modeled. The UML standard and in particular UML Profiles have already proven their effectiveness for capturing domain specific characteristics in form of models e.g. requirement models, variation models, decision models, etc. [2] [26]. MDACs can be used as a mean of capturing knowledge to be applied on these PL models as well as all required artifacts, tools and guidelines. Being platform independent, autonomous and executable, MDACs can be instantiated and sequenced in an assembly line in order to implement a PL. Thus, they can be used as a packaging unit of know-how, support for PL architecture design methodologies such as FAST [25], COPA [1] or FORM [11].

Our contribution comes in form of a two-layer MOF-compliant metamodel. The first layer, called MDAC Infrastructure, extends main UML2.0 Infrastructure classes in order to define the structure and contents of MDAC. The second layer is based on the UML2.0 Superstructure. It extends key constructs of the UML2.0 Superstructure required for the definition of the MDAC behavior independently of any language. We also discuss primary requirements for the deployment and execution of such know-how packaging unit. Let us precise that the intent of this paper is not to propose an off-the-shelf solution. We aim at introducing the basis and challenges of MDA Components. Previous efforts were done in an ITEA (Information Technology European Advancement) research project called Families [8], from which several publications where made: Bézivin et at [3] and [6] in introducing the notion of MDAC.

The paper is organized as follows; in order to avoid any confusion with Software Components, Section 2 presents how MDACs relate to them. Section 3 gives an overview of the MDAC architecture and sets some of the requirements that should be satisfied in order to support the deployment and execution of MDACs. In Section 4, we present our MOF-compliant metamodel for MDAC. It is composed of two parts, one describing its structure and its contents and the other one specifying its behavior independently of any platform. To this end, the latter makes use of UML2.0 Activity and Action constructs. Section 5 presents related works and Section 6 gives some perspectives of this work.

2 MDA Components vs. Software Component

Because the term of Component is widely used with different meanings in the software area, we find it necessary to start by clarifying how MDA Components relate to it. Grady Booch et al, define a component as "a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. Typically, it represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations" [5]. One property that MDA Components share with software components is the ability of packaging the logical units necessary for the achievement of an activity. However, unlike software components, first class citizens of MDAC are models. We mean here by model any instance -direct or indirect- of the MOF metamodel e.g. the UML metamodel and its model instances, Profiles, SPEM instance models, etc. Likewise, an MDAC packages all required artifacts for its deployment and its execution. A non exhaustive list of such artifacts could be libraries (e.g. Java, C++, etc.); guidelines; model transformation rules; configuration parameters; consistency rules in form of OCL code attached to models; icons, etc. While the principal goal of the software component discipline is to enable practical reuse of software parts and investments deprecation over multiple applications, the MDAC vision aims at enabling reuse of a certain kind of know-how applied on models. For example, in the context of the MDA, we can imagine an MDAC - one or a collaboration of MDACs - for specifying the PIM of an application domain, one for transforming that PIM to a PSM and another one for generating code from the latter. Sequencing these three MDACs then constitutes an MDAC-based production line. Of course, this would be one possible way of sequencing MDACs in order to realize an application with respect to the MDA recommendations. Considering that a company could prefer to integrate platform features at an earlier stage of the production line, an MDAC for Platform Description Model specification (PDM) would take place in earlier phases of the production line. Then, another MDAC will ensure the activity of weaving PDM outcome of the PDM MDAC and the PIM outcome of the PIM MDAC. The result of the weaving activity, i.e., a model integrating business assets as well as some platform's specific features will be one of the inputs of the model transformation MDAC and so forth.

To complete the comparison with software components, we address the notion of services required or provided by MDACs. During its execution, an MDAC provides services and may require some. Services offered by MDACs represent operations to be applied on models e.g. PIM specification, model consistency checking, model transformation, model comparisons and so on. The catalogue of services provided or required by MDAC is equivalent to software component interfaces. They represent the behavior offered by the MDAC. The interface of a software component is realized i.e. implemented by a set of operations in a specific programming language. As for MDAC, services provided might be defined either independently of any technological considerations or by using a specific language e.g. C++, Java, etc. One advantage of doing this is to keep service specifications platform independent and less vulnerable to the continuing evolution of technologies. Then, service specifications can evolve, are easily maintainable and analyzable.

3 MDA Component: An Overview

We give here an overview of the notion of MDA Component, by providing the motivation of introducing such a notion, defining in an informal way this notion and describing some features that must be supported to deploy and execute MDACs.

3.1 Rationale

In order to implement an MDA approach for a particular domain or context, one has to:

1. Define the appropriate modeling abstractions. Using the MDA related technologies, this means that specific metamodels or UML profiles are established, and that consistency checks for the models are specified.
2. Implement production automation rules that will translate a given level of abstraction into another level of abstraction, or produce some development artifact.
3. Identify reusable model artifacts that represent software concepts applicable in different projects in the same domain or technology.

Indeed, there are more activities to conduct. The MDA approach should be supported by all sorts of services in order to be easily adopted by the software developers who will need to apply it. Services such as wizards, on line help, consistency checks and connection to the platform development tools, are necessary for the good usage and acceptance of an MDA approach. Each abstraction layer, represented by a model type, must be defined with a metamodel or profile and benefits from a complete modeling environment (GUI, checks, Help, process, generators…). The MDAC is the means to reach this objective. Methodological or process related rules can also be attached to the MDAC through specific additional functionalities: Guidelines, rules, descriptions of work products to be delivered, roles participating in the usage of the MDAC are examples of such process related aspects. The result is that many kinds of elements need to be packaged together, in order to provide a complete solution for implementing an MDA approach. That single unit of packaging is called "MDA Component" (MDAC). An MDAC contains all necessary material to customize an existing modeling environment in order to apply MDA to a specific domain or context.

3.2 Definition

An MDA component is a deployable unit of packaging for the definition of types of model with dedicated tools, services and resources. The types of model can for example be a PIM or a PSM, defined in the form of a metamodel or a UML profile. The MDAC concept, in essence, is aimed at customizing an existing modeling tool, or group of tools.

It needs to be connected to the customized toolset, and to be executed accordingly in order to extend the behavior and services provided by the host toolset. When

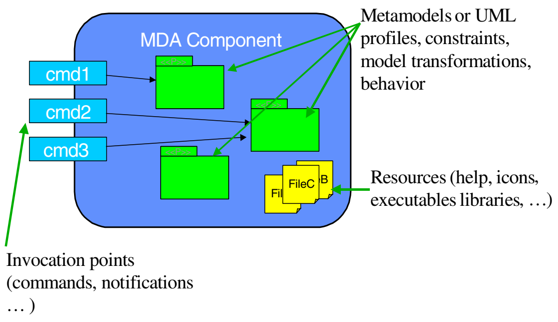

MDA Component
cmd1
cmd2
cmd3
FileC
Metamodels or UML profiles, constraints, model transformations, behavior
Resources (help, icons, executables libraries, …)
Invocation points (commands, notifications …)

Fig. 1. An MDA component packages model definitions, attached services and resources

deployed, the MDAC has to be executed in an MDA *Container*, which is a dedicated execution environment that can load the attached artifacts and descriptions, and that can execute the attached behaviors according to a predefined lifecycle. The MDA Container is frequently embedded in a hosting tool (such as a UML Case tool): it binds the hosting tool functionality to the loaded MDAC. MDACs include the definition of "*invocation points*", which abstract the different mechanisms by which the MDAC can be solicited by the environment, such as provided services, Events on which the MDAC reacts or menu entries provided to the end user. These interaction points define the connection between the MDAC's external environment (end users, other tools or MDACs) and the MDAC's implementation.

3.3 Steps from the Definition to the Usage of MDACs

The following steps and tools are necessary to support the definition, deployment and usage of MDACs:

1. *Modeling and definition of an MDAC*: In this stage, a dedicated modeling tool called *MDAC Modeler* is used to define the profiles or metamodels supported by the MDAC. The modeling support includes the modeling of profiles or metamodel or the capacity to import these definitions using XMI, the modeling or definition of the invocation points of the MDAC, the modeling or definition of the behavior, and the modeling or definition of the packaging of the MDAC.
2. *Producing and packaging an MDAC*: The definition of the MDAC needs to be checked, compiled and produced into a packaged form. The MDAC *Modeler* tool also supports this step. The result is a file in a specific format, storing the implementation of the MDAC.
3. *Deploying an MDAC:* Using the MDAC *packaging unit*, the MDAC is made accessible to a modeling tool, and can be selected by users. The targeted tools must embed an MDA *Container* to be capable of loading and executing MDACs.
4. *Applying an MDAC:* Once deployed, the users can select the MDAC in their modeling environment, and it will be executed. At this stage, the MDA *Container* loads and executes the MDAC that will customize the modeling environment and adds a new functionality. The profiles packaged by the MDAC will be applied to

the model in the user's current work context, or the metamodels packaged by the MDAC will be interpreted to type the models created under the hosting tool.

5. Maintaining existing MDACs: to make the MDACs reusable, an organization should maintain libraries of MDACs that can be applied in projects of this organization. These MDACs can evolve to support new assets, consistency checks and modeling extensions.

3.4 Runtime Support of MDACs

The MDAC execution has to be supported by a layer independent of the tool that executes it. The objective is to provide an infrastructure that is capable to be embedded within model-based tools (such as Case Tools, Meta Case Tools, or any other tool providing model oriented services) and which provides in a uniform - and hopefully standardized - way the services provided by the hosting tool. This kind of architecture is well supported by the "container pattern" vastly used by component based architectures such as CCM or EJB. Figure 2 below shows an MDAC embedded within an MDA container which isolates the MDAC from the hosting tool, allowing thus to run the same MDAC on several kinds of hosting tools.

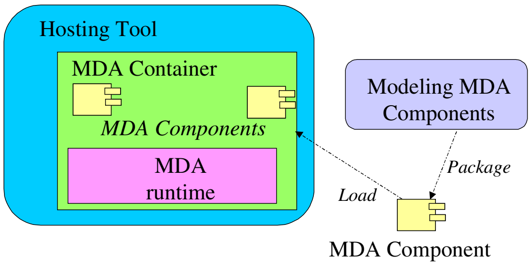


Fig. 2. Running MDACs into MDA Containers

Invocation points defined by the MDACs are the mechanisms by which the MDA *Container* will run the MDAC, and let the MDAC cooperate with the hosting tool. User interactions, events, required services can be transmitted from the hosting tool to the MDAC based on the MDA *Container* intermediation. Provided and required services can also be used to establish a dialog between several loaded MDACs. There are predefined services requested by Containers, that every MDAC shall implement, in order to let containers manage the lifecycle of the loaded MDACs. It is not expected that any hosting tool is capable of providing every requested environment and services to any MDAC. MDACs can therefore express requirements on the capacities of the hosting tools. For example, some tools are capable to interpret metamodel definitions, other support profiles for a certain version of UML, some have a strong GUI capacity, etc. It is up to the MDA *Container* to check that the hosting tool fits to the requirements expressed by the MDAC to be loaded. In the next Section, we provide a more formal definition of MDAC in the form of a metamodel.

4 The MDAC Metamodel

The definition of the MDAC we propose comes in form of a two-layers Metamodel, namely the *MDAC Infrastructure* layer and the *MDAC Behavior* layer. The *MDAC Infrastructure* layer is a MOF-compliant metamodel. It extends the UML2.0 Infrastructure concepts in order to specify the structure of the MDAC. Whilst the Behavior layer takes advantage of the recently adopted UML2.0 Superstructure constructs in order to provide a platform and language independent way for specifying the behavior of MDACs. Below we introduce both layers in more details.

4.1 The MDAC Infrastructure Layer

Figure 3 shows the part of the MDAC metamodel corresponding to the *Infrastructure* layer. A specific effort has been conducted to align this metamodel to the UML2.0 infrastructure standard. At this level, it has been deliberately decided not to use the UML2.0 superstructure metamodel, in order to stay at a model independent level. This metamodel contains the definition of an MDAC and its related classes.

MDACs are defined in terms of related model definitions, expressed as packages, dependencies on other MDACs, packaged physical units, and provided services. MDACs are a kind of NameSpace. They act as a namespace for each aggregation where the opposite role subsets “ownedmember”. The model definition is a kind of package, corresponding to the OMG metamodel definitions. These packages can be profiles, or packages representing metamodels. When the MDAC refers to standards, the model definition packages are referenced through an uri, using the OMG identification mechanism as defined in the MOF standard. MDACs are represented as an aggregation of the elements to be packaged. A Service is the formal declaration of an exposed functionality that a tool or a MDAC can provide, and that a tool or an MDAC can require. MCServices can be called internally through commands, internal calls, or event reception. Artifacts represent in general the resources or artifacts related to the MDAC. They can for example be libraries, documents, test models, user guides, icons, etc. Their exact kinds shall be defined by creating subclasses of Artifact in more concrete subPackages. These configurations are very specific to the methodology recommended for building MDAC and can vary widely. The intent behind MDAC Artifact is close to the semantics of UML2.0::SuperStructure::Artifact. Therefore, in order to match both semantics (using package merge), the name has been deliberately chosen similar.

InvocationPoint describes how an MDAC can be activated and which MDAC service is activated. It is then up to the execution container to provide the adequate mechanism to support the specified extension point and activate the service when required. An MDAC can assemble other MDACs. The assembly can be specified as a reference to another MDAC. In that case, the assembled MDAC is requested for the execution of the assembly. The assembly can be specified as “Embedded”. In that case, the assembled MDAC is packaged within the assembly one, and deployed before the assembly in the local execution space. It is expected that each language used for expressing the behavior of an MDAC has an OO structure: It can be structured by packages that own classes having operations in which the language

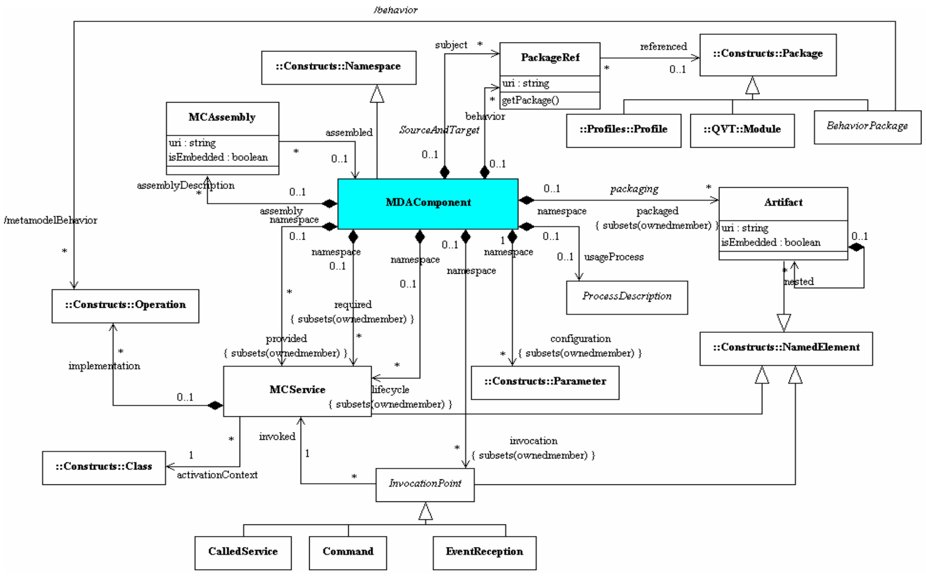


Fig. 3. A global overview of the MDAC Infrastructure layer

instructions are expressed as implementation. This is the case for QVT [17], Java, and "J"[10], but also for a vast majority of languages. As we will see below, we propose a language independent way to specify the behavior of an MDAC, but still we leave the possibility open to use a specific language, at the cost of reducing the hosting tool independence level.

4.2 The MDAC Behavior Layer

As introduced above, MDACs have a behavior that might be specified independently of any specific language. Developers will have the possibility to define the MDAC behavior either in a specific language or in an independent way. By choosing the latter solution, they will overcome the problem of the continuing technology changes and business evolutions. Indeed, when having a platform-independent behavior description, then it becomes easier to maintain, to evolve and to generate the behavior into a specific platform or language. To this end, we propose the *MDAC Behavior* layer that is a UML2.0 Superstructure-based metamodel which extends UML2.0 Activity and Action constructs by adding the properties and semantics required for the description of MDAC behaviors. Then, the "packageMerge" facility offered by the UML 2.0 standard can be used in order to merge the two layers, i.e., *MDAC Infrastructure* and *MDAC Behavior* (see figure 4). As a result, we obtain a full specification that allows the definition of both structural and behavioral aspects of MDACs. By adopting UML2.0 as a basis of the part of our metamodel devoted to the MDAC behavior description, we take advantage of:

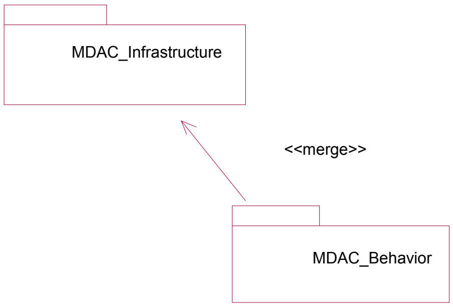


Fig. 4. The MDAC metamodel: Package Merge of MDAC layers

- o The expressiveness of the new UML2.0 for modelling executable action semantics within activities and in orchestrating them;
- o The fact that UML is currently the most widely used modeling language in the industry;
- o Tool supports and facilities;
- o Notations and diagrams offered by the standard ;
- o Easier adoption by UML modelers;

The metamodel representing the *MDAC Behavior* layer comes in form of package hierarchies. The outermost level contains two packages: the Behavior_Foundation package and the Behavior_Extensions package (see figure 5). The Behavior_Foundation package contains all UML2.0 packages required as a basis for MDAC behavior descriptions. Main ones are those related to Activities, Actions, and the Kernel package, the core of UML2.0. The Behavior_Extensions package holds classes that extend UML2.0 constructs introduced in the Behavior_Foundation package. Figure 6 points out how concepts of both packages are interconnected. It represents a global overview of the *MDAC Behavior* layer. Lighted boxes of the figure represent UML2.0 classes. Shaded boxes represent those we specified and that inherit UML2.0 classes. The main class of the Behavior_Extensions package is the MDAComponent class. An MDAComponent inherits the UML2.0::Behaviored Classifier class. A BehavioredClassifier is a Classifier that has behavior specifications defined in its namespace. One of these may specify the classifier's behavior itself which will be invoked when an instance of the BehavioredClassifier is created. One advantage is that the MDAC's behavior can be represented by state machines; this adds more control on the MDAC lifecycle. Another advantage is, being a Classifier, an MDAC can encapsulate, i.e., own other classifiers such as Artifacts as well as ActivityPerformer on these artifacts.

An Artifact is the specification of a physical piece of information that is produced, consumed, or modified by an MDAC. An MDAC provide Services and may require some. The services offered by the MDAC are realized by Operations. Operations are described in term of Activities which contain a set of Actions with an executable semantics. An Action takes a set of inputs and converts them into a set of outputs, though either or both sets may be empty. Input to, respectively, output from, an action

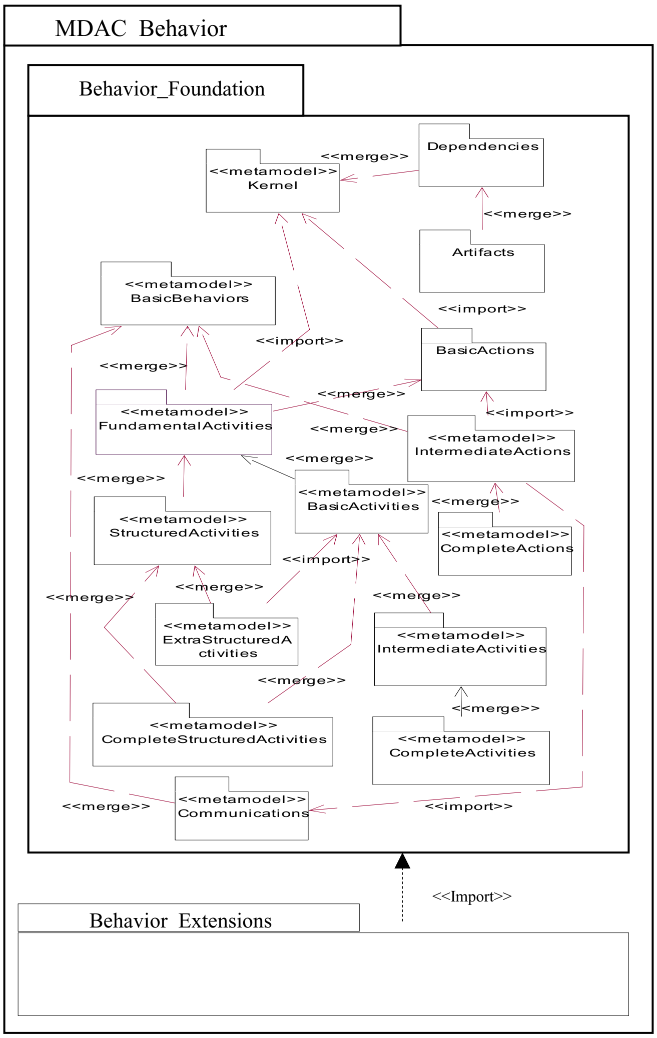

MDAC Behavior
Behavior_Foundation
<<metamodel>> Kernel
Dependencies
Artifacts
<<metamodel>> BasicBehaviors
BasicActions
<<metamodel>> FundamentalActivities
<<metamodel>> IntermediateActions
<<metamodel>> StructuredActivities
<<metamodel>> BasicActivities
<<metamodel>> CompleteActions
<<metamodel>> ExtraStructuredActivities
<<metamodel>> IntermediateActivities
<<metamodel>> CompleteStructuredActivities
<<metamodel>> CompleteActivities
<<metamodel>> Communications
<<merge>>
<<import>>
<<Import>>
Behavior Extensions

Fig. 5. MDAC Behavior layer: package hierarchies

is a typed element. It represents the Pin of the action. A Pin is typed by a Classifier. Actions consume and produce artifacts. The relation between an action and artifacts it handles is made through the fact that artifacts are Classifiers and Inputs and Outputs of an action have a type which is specified by a Classifier too. This would allow actions to manipulate artifacts as easily as calling a method while passing it parameters in usual OO programming languages. Activities have one or more ActivityPerformer who are in charge of the activity and more particularly of actions owned by it. An ActivityPerformer can be a ResponsibleRole or a SoftwareTool (e.g. compilers, model transformation engines…). A ResponsibleRole describes the rights and responsibilities of the Human who will interact with the MDAC during its execution. Indeed, due to the complexity of software development, the behavior of an MDAC can't be fully specified in terms of executable actions. Then, developer involvements may be necessary during MDAC execution. Considering this need of human interactions, we add the concepts of Interaction and Human. An Interaction is an Action and involves a Human. A Human may be an agent or a team; it has a name, a skill(s) and an authority. Finally, having in mind that an MDAC may need some tool facilities during execution-time, we decide to extend the Actions model. The CallToolServiceAction is a CallAction (see figure 7). It has InputPins which represent the arguments of the call and OutputPins as call results.

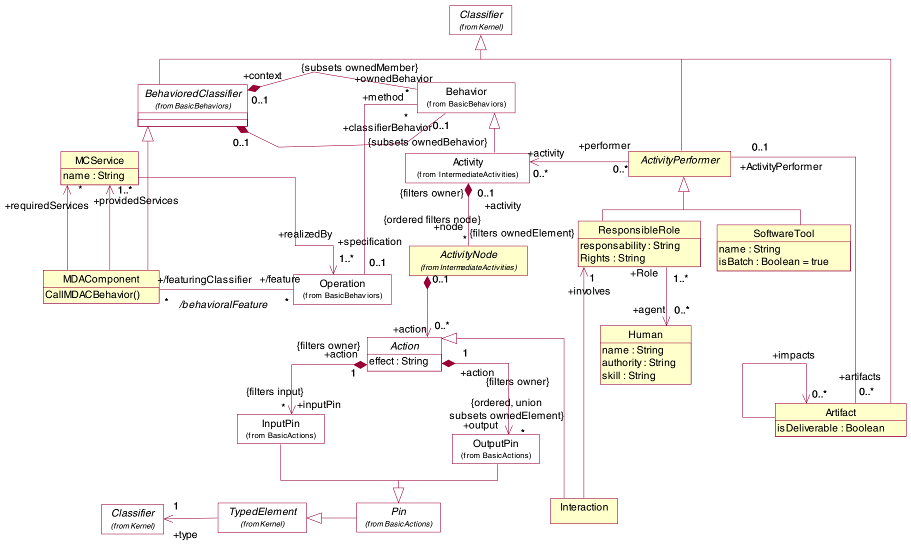


Fig. 6. A global overview of the MDAC Behavior layer

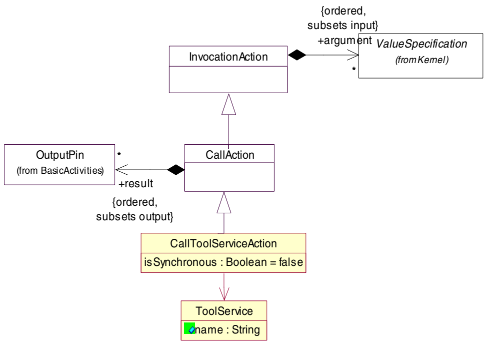

InvocationAction
{ordered, subsets input}
+argument
ValueSpecification (from Kernel)
OutputPin (from BasicActivities)
CallAction
+result
{ordered, subsets output}
CallToolServiceAction
isSynchronous : Boolean = false
ToolService
name : String

Fig. 7. The CallToolServiceAction

We make the assumption that a ToolService has a name and a set of typed parameters. One constrain on the CallToolServiceAction, would be that CallToolServiceAction arguments fit to ToolService parameters (in number and type). The model of the tool (list of services, parameters of services, binding mode…) is outside the scope of this work [4].

5 Related Works

One of the main contributions in the literature that relates to the MDA Component is the Microsoft's Software Factories vision. A Software factory (SF) is a product line that configures extensible development tools e.g. Microsoft Visual Studio Team System (VSTS) with packaged content and guidance, carefully designed for building specific kinds of applications [9]. Main concepts in SF are the software factory schema and the software factory template. A SF schema lists artifacts like source code, SQL files, etc. and how they should be combined to create a product. It specifies which Domain Specific Languages (DSLs) should be used and defines the product line architecture. As for SF template, it packages all the artifacts described in the SF schema. It provides patterns, guidance, templates, DSL editing tools, etc. used to build the product. Then, an extensible development environment will be configured by the SF template in order to become a SF for a product family.

Looking at these definitions of SF, one can deduce that Microsoft's Software Factories and MDA Components are competitors. Indeed, SF and MDAC promote the same vision of reusing software skills for the sake of a more cost-effective software production and short time-to-market. However, when MDAC first class entities are MOF instance models, SF promotes the use of DSLs. This is, in our opinion, a delicate issue. In [9], authors argued that in some cases, UML profiles or MOF are not well suited for modelling some business concepts which are more naturally expressed in a specialized syntax. They also support that DSLs are easy to create, to evolve and can be used to implement solutions based on these DSL. We don't agree with this vision. In our case, the primary reason why we chose a standard way (i.e. MOF instances such like UML) for representing models is to be independent of any platform or language. Using a DSL for making models more expressive is not a problem in itself. The problem is that it requires that the DSL semantics be understandable by tools and project stakeholders. Then, even if tools are compatible within the same SF template (package), how do we deal when these tools need to exchange models with other tools using different DSLs? Moreover, if the behavior of a software factory is defined through thousands of code source lines, how the SF customer could maintain or customize the SF? We believe that mechanisms like stereotypes and tagged values offered by UML profiles as well as OCL (Object Constraint Language) [24] are expressive enough to capture the characteristics of a specific domain context. The OMG already provides Profiles for Software Process Engineering (SPEM), System engineering, test modeling, QOS modeling, and the already long existing list of profiles will proliferate within the coming years [18]. For those who may find limitations of using UML profile, MOF can be used as formalism for defining domain specific language metamodels. This allows leveraging UML standard tools and training. Thus even if SF and MDACs share the same vision of promoting software production lines (see Section 1.1), they do not follow the same approach. However, MDA Components can be used by Software Factories as a standard mean for representing certain knowledge and for packaging required artifacts and tools for applying it.

6 Conclusion

In this paper, we introduced the notion of MDA Component, i.e., a packaging unit for encapsulating business know-how and required resources in order to support specific operations on a certain kind of model. A standard formalism for representing MDA Components was proposed in form of MOF-compliant metamodel and requirements for their deployment and execution was established Most of the work which has contributed to this paper has been done within the Modelware IST project [14]. Tooling and Use Cases are currently underway, a first implementation being already done. In parallel, there is an OMG effort to standardize the notion of MDA component. Currently the writing of an RFP is underway. We believe that the MDA component notion, once standardized and tooled will provide its full power to the MDA technique, in order to capitalize, share, reuse, improve and automatically apply software development know-how. Still there is work to do in MDAC definition. For example, the MDA Container "universal" architecture is under study, the mechanisms for combining MDACs have to be finalized, and the level of tool independence of MDACs has to be formalized into predefined compliance levels. The highest challenge is to meet a standard definition that ensures a complete tool independence.

References

[1] America P., Obbink H., Muller J., and van Ommering R. "COPA: A Component-Oriented Platform Architecting Method for Families of Software Intensive Electronic Products", in Proc. of the 1st. Conf. on Software Product Line Engineering, Denver, Col, USA, 2000.

[2] Anastasopoulos M., Atkinson C. and Muthig D. "A Concrete Method for Developing and Applying Product Line Architectures" in Proc. of the Net.Object Days, Erfurt, Germany,2002, LNCS, Springer, Vol. 2591 / 2003

[3] Bézivin J., Gérard S., Muller P-A. and Rioux L. "MDA components: Challenges and Opportunities", in: Metamodelling for MDA, York, England, 2003.

[4] Blanc X., Gervais M.P., and Sriplakich P. "Model Bus: Towards the Interoperability of Modelling Tools", in Proc. of the MDAFA'04, Linköping University, Sweden, 2004.

[5] Booch G., Rumbaugh J. and Jacobson I. "The Unified Modeling Language User Guide", Addison-Wesley Professional; 2 edition, July 19, 2004.

[6] Desfray P. "Techniques for the early definition of MDA artifacts in a UML based development" Enterprise UML & MDA, London May 12 and 13 at: http://www.enterpriseconferences.co.uk/programme.pdf

[7] EDOC, "UML Profile for Enterprise Distributed Object Computing", OMG Document ptc/02-02-05, 2002 http://www.omg.org.

[8] Families ITEA Project at: http://www.esi.es/en/Projects/Families/

[9] Greenfield J., Short K. "Software factories: assembling applications with patterns, models, frameworks and tools", in Proc. of the 18th Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), Anheim, CA, USA, 2003, ACM press.

[10] J language, at: _ http://www.objecteering.com/pdf/whitepapers/us/uml_profiles.pdf

[11] Kang K. C., Kim S., Lee J., Kim, Shin E., and Huh, K M. "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures", Annals of Software Engineering, Vol. 5, 1998.

[12] MDA Development tools, at: http://www.omg.org/mda/committed-products.htm

[13] MDA Guide. "Model Driven Architecture (MDA)", OMG TC document ormsc/2001-07-01, July 2001, at http://www.omg.org.

[14] MODELWARE Project, at http://www.modelware-ist.org

[15] MOF 1.4. "Meta-Object Facility", OMG document formal/2002-04-03, April 2002, at http://www.omg.org.

[16] Muthig D., Atkinson C. "Model-Driven Product Line Architectures" in Proc. of the 2nd International Software Product Line Conference, San Diego, CA, USA, 2002, LNCS,. Springer, Vol. 2379/2002

[17] OMG, Request for Proposal MOF2.0 Query /Views/Transformations, OMG document: ad/2002-04-10, April, 2002, at http://www.omg.org.

[18] OMG specifications at: http://www.omg.org/technology/documents/modeling_spec_catalog.htm

[19] Perry, D.E., "Generic Architecture Descriptions for Product Lines", ARESII: Software Architectures for Product Families, Los Palmos, Gran Canaria, Spain, 1998, LNCS,. Springer, Vol. 1429/1998.

[20] SPEM1.1, "Software Process Engineering Metamodel", OMG document formal/02-11/14, November 2002, at http://www.omg.org.

[21] Standish Group: "2004 Third Quarter Research Report" at: http://www.standishgroup.com. Page last visit: June 13, 2005.

[22] UML2.0 Infrastructure, "Unified Modelling Language", Final Adopted Specification, OMG document ptc/03-09-15, December 2003, at http://www.omg.org.

[23] UML2.0 Superstructure, "Unified Modelling Language", Available Specification, OMG document ptc/04-10-02, October 2004, at http://www.omg.org.

[24] UML2.0 OCL Specification, "Unified Modelling Language 2.0 Object Constraint Language", Adopted Specification, OMG document formal/03-10-14, October 2003, at http://www.omg.org.

[25] Weiss D., Lai C., and Tau R., "Software product-line engineering: a family-based software development process", Addison-Wesley, Reading, MA, 1999.

[26] Ziadi T., Hélouët L., Jézéquel J-M. "Towards a UML Profile for Software Product Lines", in Proc. of International Workshop on Product Family Engineering (PFE-5), Seana, Italy, 2003, Springer LNCS 3014/2003, P. 129-139.

An MDA Approach for Adaptable Components

Steffen Göbel

Institute for System Architecture,
Dresden University of Technology, Germany
goebel@rn.inf.tu-dresden.de

Abstract. Components should provide maximum flexibility to increase their reusability in different applications and to work under changing environment conditions as part of a single application. Thus, adaptation and reconfiguration mechanisms of single components and component assemblies play a crucial role in achieving this goal. In this paper we present a model of Adaptable Components that allows modelling of adaptation and reconfiguration operations taking place at development, deployment or runtime. The concept of composite components is utilized to encapsulate adaptation operators and to map component parameters to different predefined internal configurations of subcomponents. The component model is not tied to a particular component platform. Instead, it can be mapped to existing component platforms like EJB using an MDA approach. Different Platform-Specific Models for the same target component platform enable tailored flexibility for particular component deployments. For example, a model can support or not support runtime reconfiguration. Extensions to UML diagrams are introduced to graphically model reconfiguration operations.

1 Introduction

Two of the key success factors of component-based software engineering are *reuse* of components in different applications and *composition* of complex applications out of well-defined and loosely coupled parts. Component platform standards like JavaBeans, Enterprise JavaBeans (EJB), Microsoft COM, and Microsoft .NET have crucially contributed to this success. To increase reusability and flexibility, components should support mechanisms to adapt them to different environments and to add new functionality.

This adaptation process can be carried out at different times in the component life cycle: at development time, at deployment time, or at runtime. However, components look quite differently and consist of different artefacts at these three times. For example, at development time a component might comprise a set of Java source files and an XML descriptor. At runtime the same component might consist of binary program code structured in classes and several data structures. As a result, the conceivable adaptation mechanisms also differ at the three times. Some mechanisms might change the source code of components and others might employ interceptors at runtime.

A variety of adaptation mechanisms has been developed by the research community and software industry, but most of them are only applicable at one time, either development, deployment, or runtime. A unified model describing adaptable components at all three times is missing. In this paper we introduce such a model for the development of Adaptable Components (ACs). It employs the concept of composite components to encapsulate adaptation mechanisms. *Adaptation* is considered as a superset of the concepts *reconfiguration*, *customization*, and *parameterization* in this paper.

The model of ACs combines the following ideas: (i) The adaptivity is expressed by *component parameters*. The notion of component parameters comprises component properties used, for example, in JavaBeans and type parameter used in C++ templates or Java generics. In a previous work [7] we also showed that QoS properties are also a kind of parameters. (ii) An AC as a composite component encapsulates a set of predefined internal configurations of subcomponents. Component parameters are mapped to these configurations. That means that the internal configuration of an AC changes if component parameters are changed. (iii) The model of ACs is platform and programming language independent. An MDA approach enables transforming of the model to different target component platforms. Currently, only Java is supported as programming language. (iv) Existing adaptation mechanisms can be integrated by special modelling operators.

The paper is structured as follows: In the next section we introduce our model of ACs, explain model constituents, and modelling techniques. In the third section the model is illustrated based on a simple example—a crypto component. The fourth section deals with the implementation of the model and, in particular, with the MDA approach to transform the model to different component platforms. The paper closes with an examination of related work, a conclusion, and an outlook to future work.

2 Model of Adaptable Components

This section first describes the constituents and concepts of our model of ACs. The concept of composite components is employed to encapsulate model constituents, especially subcomponents. The advantage of this approach is that ACs do not look and behave differently than other components from outside. Many constituents (e. g., adaptation operators and glue code) in the model are optional, which increases flexibility and enables several application scenarios.

The remaining part of this section deals with mapping of component parameters and modelling of configurations and reconfigurations by using UML diagrams. New UML stereotypes are introduced to describe structural reconfiguration operations.

2.1 Component Constituents

The model of ACs is schematically depicted in Fig. 1. ACs define explicit dependencies to other components by fixed sets of typed *provided* and *required* ports each implementing a certain interface. Any communication with other components requires explicitly defined connections to these components via ports. The ports of an AC stay the same, even if the internal configuration is changed in the course of a reconfiguration. This means that the adaptation process stays transparent to other components of an application. A *management interface* (similar to the *equivalent interface* in CORBA Components) is used to navigate to required and provided interfaces and access the component parameter interface. The *parameter interface* provides read and write access to component value parameters (properties) by `get` and `set` methods.

We define the *configuration* of an AC as a set of subcomponents, a set of adaptation and aspect operators applied to subcomponents, a set of connections between ports of these subcomponents, and a binding between the external ports of the AC and ports of subcomponents. Thus, the *active configuration* determines the current runtime properties of a component, a kind of operating range. The set of valid configurations is defined by the *adaptation specification* containing the initial configuration and transitions to other valid configurations (see Sec. 2.2 for details).

The *component repository* contains all available subcomponents of an AC. A subset or all of these subcomponents, which are designated as *active subcomponents*, form an active internal configuration at runtime. Only active subcomponents process incoming method calls and the active configuration determines communication links. The *management interface* enables adding of new subcomponents and accordantly configurations at runtime. The *interface binding* as the central hub forwards requests of external ports to ports of active subcomponents and also connects internal subcomponents. The *adaptation manager* controls the interface binding and the reconfigurations using the adaptation specifica-

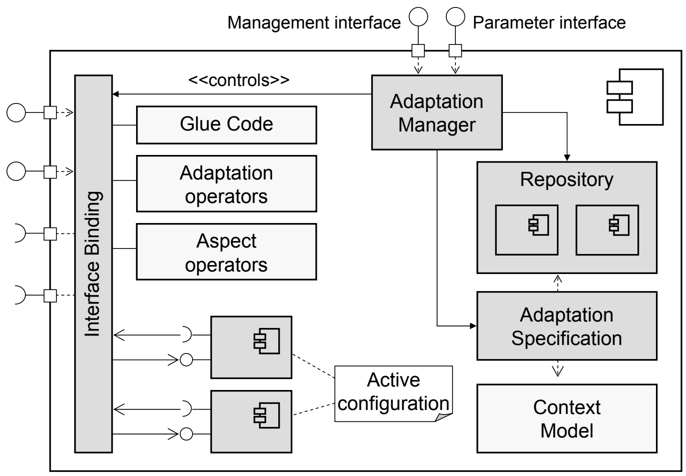


Fig. 1. Model of a adaptable component

tion. It thereby enforces consistency constraints and synchronizes reconfiguration operations.

Glue code is a special kind of subcomponent. The difference is that subcomponents are usually reused or at least intended to be reused by other applications whereas glue code is tailored to a particular AC. Glue code enables adding missing functionality to an AC (e. g., a new interaction protocol) that otherwise prevents subcomponents from working together.

Adaptation operators take a single subcomponent as input and transform it to a new subcomponent with a possibly different set of ports. They are typically applied to adapt incompatible port interfaces. Incompatibilities occur because component originally developed by different developers and for different applications are reused in a new application. For example, incompatibilities are caused by slightly different method names, different method signatures, and missing methods. Potential implementation approaches for adaptation operators include, but are not limited to:

– adapters deployed at runtime that extend or change some functionality of a subcomponent with or without providing a new interface;
– byte code transformers [9] that adapt subcomponents at deployment or loading time and before the application is started;
– source code transformers [2] that modify the source code of subcomponents before or during the compilation.

The actual implementation strategy is left open by our model.

Aspect operators are applied to a set of subcomponents and influence the behaviour at defined joinpoints like in aspect-oriented programming [10]. As with adaptation operators, different types of aspect operators can be employed at development, deployment, and runtime. Joinpoints include, but are not limited to, places before and after methods, component instantiations, component destructions, and reconfiguration operations.

So far we explained that the management interface of AC is used to change parameters and therewith the active internal configuration. However, these parameter changes need not necessarily be triggered by an external entity but can also be triggered by the AC itself, for example, if environment conditions change. The adaptation manager can utilize a *context model* of the environment to check conditions defined by the adaptation specification and change component parameters if specified. This way self-adaptable components can be modelled utilizing the mechanism of parameter mapping.

2.2 Modelling of Configurations and Configuration Variations

The modelling process of ACs defines the *adaptation specification* as described in the last subsection and thus all valid configurations of an AC. Two different approaches support the graphical specification of valid configurations: *complete configurations* and *configuration variations.*

Complete configuration—as the name suggests—graphically describe all subcomponents, connections, interface bindings, and parameter settings of a particular configuration by an UML component diagram. UML 2.0 supports all required concepts including the description of interface bindings using the delegate construct from the composite structure package. The description of complete configurations for ACs is useful if only a small set of different configurations exists and if these configurations differ in many aspects (e. g. different subcomponents and connections).

However, modelling of complete configurations is not very flexible if several configurations of an AC have only a few structural differences. For example, it should be possible to model an AC where an integer parameter specifies the number of instances of a particular subcomponent. Instead of modelling many complete configurations with different numbers of this subcomponent, it is sufficient to model the initial configuration with one subcomponent and the *reconfiguration operations* that add one subcomponent. Applying these reconfiguration operations multiple times to the initial configuration can create all desired configurations.

Thus, configuration variations as the second model approach describe changes applied to a particular initial configuration to indirectly define a new configuration. Six different atomic reconfiguration operations can be identified to describe a transformation from one configuration to another one:

– changing component parameter
– adding a connection between subcomponents or between an external interface and a subcomponent's interface
– removing a connection between subcomponents or between an external interface and a subcomponent's interface
– adding a subcomponent
– removing a subcomponent
– replacing a subcomponent by another subcomponent

It can easily be proven that these reconfiguration operations are sufficient to transfer any configuration to any other configuration[1].

UML has no built-in mechanisms to model reconfigurations of component nets, but the UML meta-model can be extended using stereotypes or new meta-classes for new concepts. Thus, we defined extensions for each of the above mentioned reconfiguration operations. New connections and subcomponents are highlighted and mark with a plus symbol as stereotype. Removed connections and components are drawn with a dashed line and a minus symbol as stereotype. Fig. 2 depicts the graphical notation of the extensions. These extensions can now be utilized to graphically model configuration variation consisting of several reconfiguration operations with UML diagrams. The developer takes a component diagram of a particular configuration as starting point and then graphically models the reconfiguration operations to get to the new configuration.

[1] A trivial transition first removes all connections and subcomponents of the old configuration and adds all new subcomponents and connections later on.

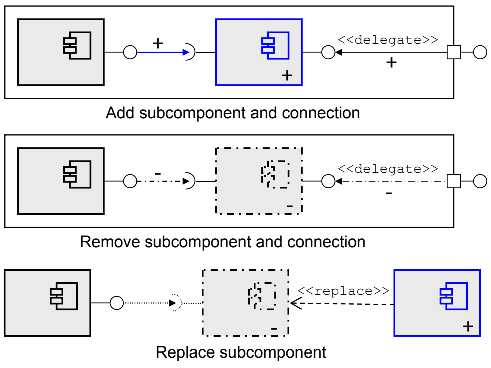

<<delegate>>
Add subcomponent and connection
<<delegate>>
Remove subcomponent and connection
<<replace>>
Replace subcomponent

Fig. 2. UML extensions for reconfiguration modelling

2.3 Mapping of Component Parameters

In general, the parameter mapping is defined by a function with component parameters as arguments that selects a predefined configuration and an ordered set of configuration variations applied to this configuration. Additionally, a parameter can influence parameters of subcomponents using the *change parameter* reconfiguration operation.

The setting of component parameters, which then selects the active configuration, is considered at four different times (see also Sec. 4.3):

Development time. The developer can optionally define a default value for a parameter. It can be overridden at deployment time or runtime.

Deployment time. The parameter value is set during the deployment of the application.

Creation time. The parameter value is set during creation of a new component instance.

Runtime. The parameter value is changed at an already existing component instance during the runtime of the application. This process is generally called *reconfiguration*.

Development, deployment, and creation time parameter settings can be supported quite easily. The components just need to be initialized and wired according to the resulting configuration. However, changing parameters at runtime, which possibly leads to a reconfiguration of components, requires considerably more effort. Particularly, the runtime environment of components has to address the following issues that can also be found in database transactions: *atomicity* – a reconfiguration must be executed completely or not at all; *consistency* – an AC must be in a valid state before and after a reconfiguration transition; and *isolation* – a reconfiguration must be executed without the interference of other reconfigurations or application activities.

Two kinds of parameters are considered: parameters with a finite set of possible values (*enumeration*) and parameters with integer or real values in a certain interval. In the following, we describe a few special forms of parameter mapping.

– The simplest form of parameter mapping assigns a particular configuration to each enumeration parameter value. For example, a component parameter "compressionAlgorithm" with the possible values "bzip2" and "gzip" could be mapped to two different configurations, respectively.
– A configuration is assigned to an interval of integer or real parameters. For example, a component parameter "compressionLevel" with a allowed range from 0 to 10 could be mapped to two different configurations in the intervals 0–5 and 5–10.
– Parameters are directly or indirectly mapped to parameters of subcomponents.
– An integer parameter specifies how often a configuration variation should be applied to a configuration.

3 Example: Crypto Component

In this section a crypto component is taken as a simple example to illustrate the use of ACs. The crypto component shall support a set of symmetric encryption algorithms (AES, Twofish, Blowfish). The interface contains only two methods: `encrypt` and `decrypt`. They enable encrypting and decrypting of given byte arrays with a given encryption key. Different loss-less compression algorithms (deflate, bzip2) are also part of the crypto component to optionally compress data before encryption and decompress it after decryption, respectively. Data compression is useful to reduce data size and to maximize the entropy of data, which complicates possible attacks on encryption algorithm.

Several parameters are exposed to adapt the crypto component:

cryptoAlgorithm is an enumeration parameter to select the active encryption/decryption algorithm (values: AES, TWOFISH, BLOWFISH).
key is an byte array parameter to set the active encryption key. This parameter does not influence the configuration of the crypto component.
compressionAlgorithm is an enumeration parameter to select the active data compression algorithm (values: DEFLATE, BZIP2, NONE).

If a user of the crypto component changes these parameters, a different internal configuration of the crypto component as depicted in Fig. 3 is selected.

The crypto component consists of three subcomponents for encryption algorithms and two subcomponents for compression algorithms. We assume that these subcomponents are available as libraries (e. g., open source projects) and need not to be implemented. Glue code implements the coupling of encryption/decryption and compression/decompression because this functionality was not supported by the crypto algorithms. All it does is calling the compression method of a compression subcomponent before calling the encryption method of a encryption subcomponent and accordingly for decompression and decryption. An adaptation operator is required for the Twofish subcomponent because it has a slightly different and therefore incompatible interface than the other encryption components. The methods for encrypting and decrypting are named `encryptBytes` and `decryptBytes`. The adaptation operator internally maps the different methods pairs to change the interface to be compatible with the other subcomponents.

The crypto component is intended to be reused in different applications. For example, the following use cases are conceivable:

- The crypto component is used in a application with a fixed configuration, for example, always with AES and Inflate. This means that all parameters except the encryption key are set at development or deployment time.
- The parameters of the crypto component are set when creating new instances. However, once the instances have been created, parameters are immutable.
- The parameters of the crypto component are changeable at any time to give maximum flexibility.

Each of these scenarios enables different optimizations (see Sec. 4.3) but this should be completely transparent to the application developer. For example, in the first scenario unused subcomponents can be omitted whereas the last scenario requires additional program code to handle runtime reconfiguration.

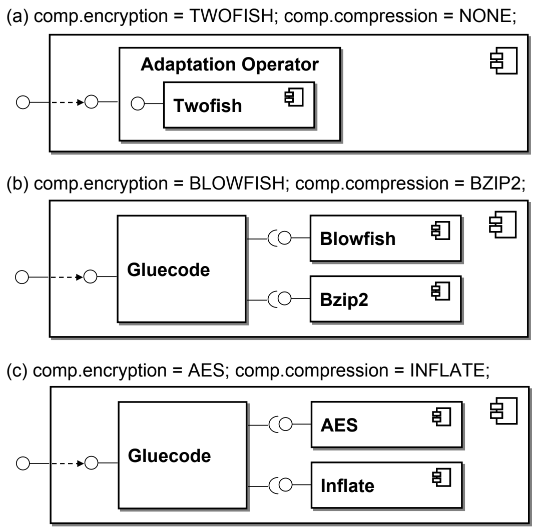

(a) comp.encryption = TWOFISH; comp.compression = NONE;
Adaptation Operator
Twofish
(b) comp.encryption = BLOWFISH; comp.compression = BZIP2;
Gluecode
Blowfish
Bzip2
(c) comp.encryption = AES; comp.compression = INFLATE;
Gluecode
AES
Inflate

Fig. 3. Example: Crypto component with different internal configurations

4 Implementation and Mapping to Component Platforms

Our model of ACs presented in Sec. 2 is independent of any component platform and does not define any implementation details deliberately. However, the model is useless without an actual implementation. There are at least two implementation strategies: (i) all model constituents and concepts are directly supported by the component platform and associated framework or (ii) all model constituents are mapped to an existing component platform (e. g., EJB or JavaBeans) and missing functionality is either generated or implemented as auxiliary services. In this paper we only describe the latter approach but in a previous work [8] we also showed the feasibility of the first approach. The main advantage of the mapping approach is that existing application servers and component platforms can be reused. It is also feasible to integrate a single new AC in an existing component-based application with reasonable overhead. In this way expensive developments from scratch can be avoided.

In this section we first describe the implementation of the metamodel of ACs and a generic framework to support the life-cycle of ACs, especially creating of instances, parameter mapping, and runtime reconfiguration. Next, we show how the model can be mapped to a component platform in general, and to EJB in particular. We explain how the different model concepts can be emulated by the target component platform and which constraints are necessary for the development of components. Other component platforms might be supported in a similar way.

4.1 Metamodel of ACs

The metamodel of an AC contains all the information necessary to control AC's life cycle—from creation, over reconfiguration until destruction. It describes all subcomponents, adaptation operators, interfaces, configurations, variations, etc. We chose the Eclipse Modeling Framework (EMF, [6]) as tool for implementing the metamodel. EMF provides a lot of useful features that simplified our implementation. It employs a subset of MOF 2.0 called Ecore to store metamodels, it generates code to access instances of the metamodel (in our case AC models), it generates a simple Eclipse Plugin as editor for the metamodel, and it includes default support for persistence of model instances using XMI.

Fig. 4 depicts a simplified class diagram of AC's metamodel. `AdaptableComponent` is the entry point to this diagram. An AC has a set of `Configurations` that are mapped to certain parameter values or value ranges via `ParameterMappings`. A `Configuration` defines the set of `ComponentInstances` that are wired using several `Connections` between component `Ports`. A `Reconfiguration` defines necessary `ReconfigurationOperations` to switch from one configuration to another one.

The metamodel is not only used by runtime support of ACs (see next section) but also by the modelling tool to design and develop ACs. This modelling tool is currently under development and will be realized as an Eclipse Plugin.

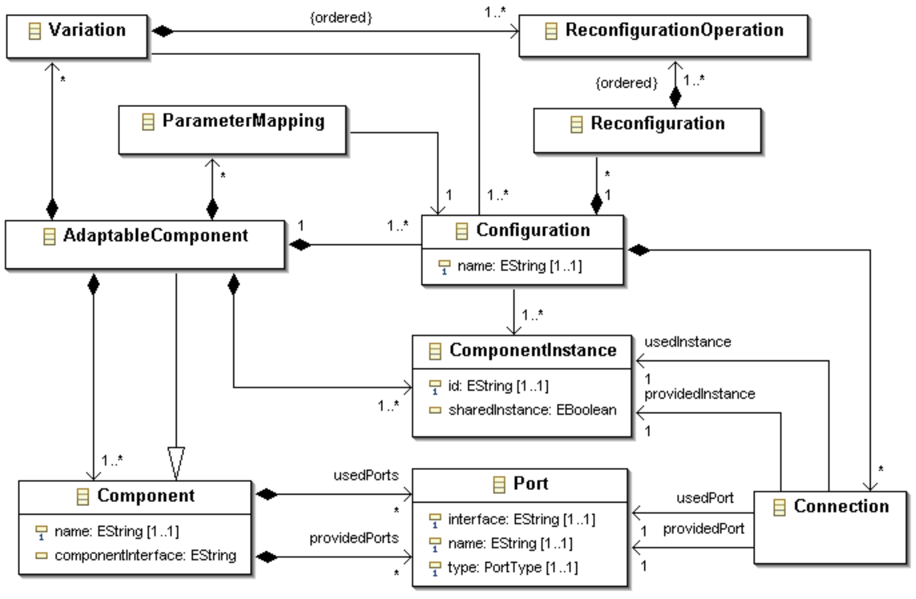


Fig. 4. UML class diagram of AC's metamodel

4.2 Generic Life-Cycle Support for ACs

Although the mapping to different component platforms differs in many details, we have developed a generic framework that can be used and extended by platform-specific implementations. It contains a set of Java classes implementing common functionality for managing AC's life cycle and relies on the implementation of the metamodel. It is employed to develop platform-specific adaptation managers, component repositories, and interface bindings (see Fig. 1). The offered functionality can be divided into three categories: creation of instances, runtime reconfiguration, and helper functions.

Creation of Instances. Several steps are necessary to create an AC instance. First, the current values of component parameters are used to determine the active configuration of the AC. Next, all subcomponents including glue code and adaptation operators are instantiated according to the configuration. In the last step, ports of subcomponents are connected among each other and with external ports of the AC. These steps are not necessarily all performed at runtime of the AC. For example, the first step might also be performed during deployment of an AC and could generate appropriate code. The actual implementation strategy is defined by the platform-specific model (see Sec. 4.3).

Runtime Reconfiguration. Parameter changes at runtime can trigger reconfigurations of ACs. A reconfiguration must adhere to the characteristics described in Sec. 2.2 and is performed in several steps. First, the new configuration is determined by the parameter mapping. Next, all component instances that are influenced by reconfiguration operations (e. g., an instance might be removed) must be stopped putting the component in an inactive state. Inactive means that no method call is currently in progress in this instance and new calls are blocked. Special caution is required to prevent deadlocks in this step. Next, all necessary reconfiguration operations are executed. In the last step, all blocks are removed and method calls can be continued.

Helper Function. Several functions (e. g., creating and removing connections and subcomponents) are provided to solve small tasks both during the creation of instances and the runtime reconfiguration. Many of them are intended to be implemented or extended by platform-specific code. Thus, they are hooks for platform-specific behaviour.

4.3 Model Mapping

The mapping of our model of ACs to a particular component platform requires that all model constituents and concepts described in Sec. 2 are supported or emulated by means of the target platform. The most important concepts to be mapped are composite components, explicit dependencies by ports, and component parameters. This leads to a Platform-Specific Model (PSM) and a platform-specific implementation, later on.

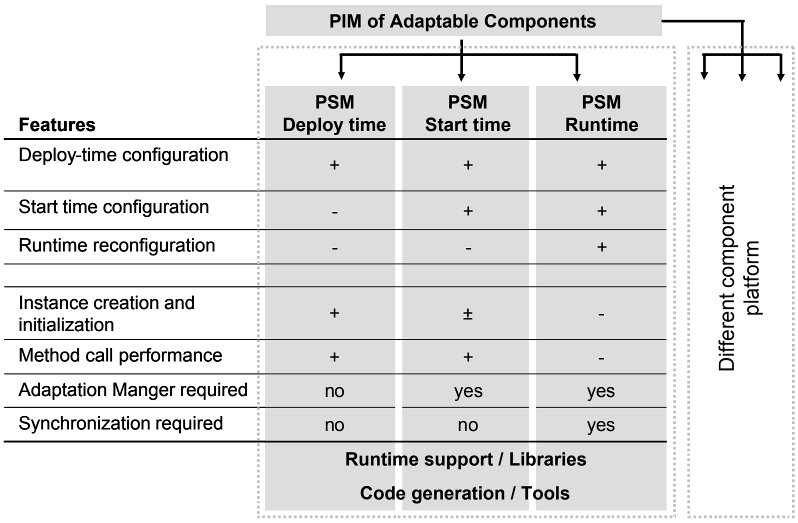
The emulation of unsupported features by the target component platform often requires defining constraints. That means that certain feature of the component platform must not be employed for developing ACs. For example, many component platform use name servers to acquire references to collaborating components and the accordant program code is tangled in the business logic. Using this feature could violate explicitly defined dependencies between components.

We consider at least three different PSMs for each component platform. The different features are illustrated in Fig. 5. The right level of flexibility for a certain AC can be chosen depending on application needs. This choice requires no changes to the AC by the developer. It is all transparently managed by the runtime framework and development tools.

4.4 Example: Model Mapping to EJB

We chose EJB as target component platform to demonstrate how a model mapping can be achieved. As described in the previous section, each model constituent and concept must be emulated by features of EJB. Due to limited space we only focus on the "PSM Runtime“ according to Fig. 5. Since runtime reconfiguration is the most complicated part, the other two PSMs are even simpler to realize.

EJB does not directly support composite components, which are a key concept in our component model to encapsulate all other constituents. However, it


PIM of Adaptable Components
Different component platform
Runtime support / Libraries
Code generation / Tools

Features	PSM Deploy time	PSM Start time	PSM Runtime
Deploy-time configuration	+	+	+
Start time configuration	-	+	+
Runtime reconfiguration	-	-	+
Instance creation and initialization	+	±	-
Method call performance	+	+	-
Adaptation Manger required	no	yes	yes
Synchronization required	no	no	yes

Fig. 5. Overview and comparison of the model mapping

is feasible to flatten ACs and emulate this concept with functionality available in EJB. We defined the following mapping rules and constraints:

– A single stateful session bean is generated for every AC and its home interface is bound to JNDI (Java Naming and Directory Interface). This session bean also implements the management and parameter interface. Method calls are forwarded to the adaptation manager.
– Every subcomponent is bound to JNDI with unique names only known to the AC session bean.
– A `get`*PortName* method is generated for every subcomponent. It hides the process of getting references by JNDI from the component logic. The Adaptation Manager initializes all references during start-up or after a reconfiguration.
– Business methods of subcomponents must not directly use JNDI but the generated helper methods to access other components. This constraint is necessary to enable explicitly defined connection between components and reconfiguration at runtime.
– Glue code is encapsulated in a session bean and otherwise handled like any other subcomponent.

Other components outside the AC only work with the generated session bean like with any other session bean and need not to be aware of implementation details.

The support of runtime reconfiguration of ACs with EJB is relatively easy to implement. The EJB specification enforces non-reentrant instances. The EJB container must ensure that only one thread can be executing an instance at any time. That means that a method call and a parameter change possibly triggering a reconfiguration can never happen at the same time. Thus, no additional code needs to be injected to block method calls during a reconfiguration.

5 Related Work

Many Architecture Description Languages (ADL, [12]) use composite component concepts to provide a uniform view of applications at different abstraction levels. However, ADLs focus on highly distributed systems and many view configurations statically. Exceptions are Darwin [5] and Rapide [11] supporting constrained changes of configurations as well as C2 [14] and Wright [1] supporting almost arbitrary configuration changes at runtime. As opposite to our approach, reconfigurations are specified by program code or scripts, a model transformation to different component platforms is not considered, and a mapping of component parameters to different configurations is not supported.

OpenORB and OpenCOM [4,3] aim at developing a configurable and reconfigurable component-based middleware platform. They support composite components based on enhancements of Microsoft COM and, especially, offers extensive meta-programming interfaces to adapt components and the middleware itself at runtime. ObjectWeb Fractal [13] completely focuses on the development of an adaptable composite component model with reflection and introspection capabilities to monitor a running system. Components need not to adhere to a fixed specification, but can rather choose from an extensible set of concepts and corresponding APIs depending on what they can or want to offer to other components. However, both approaches focus on the definition of appropriate interfaces and not on the design of adaptable components. Reconfigurations must be developed by program code using meta-programming APIs. The reconfiguration mechanisms are integrated in a special-purpose component platform.

6 Conclusions and Outlook

In this paper we presented a model of adaptable components that is independent of a particular component platform. It enables mapping of component parameters to different internal configurations and, therewith, encapsulating of structural adaptation. Adaptation operators and glue code as optional model constituents help to integrate and reuse existing components in new ACs. Different PSMs for a single target component platform enable tailored flexibility according to application needs.

We showed how this model can be mapped to the EJB platform as an example. Additionally, we have already implemented support for JavaBeans. Mappings to WebServices and JMX are currently under development. We also work on sophisticated tool support for designing and developing ACs.

Some detail problems will be addressed by further research: In some cases, the state of components must be transferred to replacement components in the course of a reconfiguration. This will be supported by special state transform operators. We will also investigate how the generated code, especially to synchronize reconfigurations, can be optimized.

References

1. R. Allen, R. Douence, and D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, 1998.
2. U. Aßmann. *Invasive Software Composition.* Springer-Verlag, 2003.
3. G. S. Blair, G. Coulson, L. Blair, H. A. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas. Reflection, Self-Awareness and Self-Healing in OpenORB. In *Workshop on Self-Healing Systems (WOSS '02)*, pages 9–14, Charleston, SC, USA, 2002.
4. G. Coulson, G. S. Blair, M. Clarke, and N. Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126, 2002.
5. S. Crane, N. Dulay, J. Kramer, J. Magee, M. Sloman, and K. Twidle. Configuration management for distributed software services. In *IFIP/IEEE International Symposium on Integrated Network Management (ISINM'95)*, Santa Babara, USA, 1995.
6. Eclipse Modeling Framework (EMF). http://eclipse.org/emf/.
7. S. Göbel. Encapsulation of Structural Adaptation by Composite Components. In *ACM SIGSOFT Workshop on Self-Managed Systems (WOSS'04)*, Newport Beach, CA, USA, 2004.
8. S. Göbel and M. Nestler. Composite Component Support for EJB. In *Winter International Symposium on Information and Communication Technologies (WISICT'04)*, Cancun, Mexico, 2004.
9. R. Keller and U. Hölzle. Binary Component Adaptation. In *ECOOP 09*, Brussel, 1998. Springer.
10. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th European Conf. on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.
11. D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.
12. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *Transactions on Software Engineering*, 26(1):70–93, 2000.
13. ObjectWeb. Fractal. http://fractal.objectweb.org/.
14. P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

Layered Ontological Modelling for Web Service-Oriented Model-Driven Architecture

Claus Pahl

Dublin City University, School of Computing, Dublin 9, Ireland
cpahl@computing.dcu.ie

Abstract. Modelling is recognised as an essential activity in the architectural design of software systems. Model-driven architecture (MDA) is a framework implementing this idea. Ontologies are knowledge representation frameworks that are ideally suited to support modelling in this endeavour. We propose here a layered ontological framework that addresses domain modelling, architectural modelling, and interoperability aspects in the development of service-based software systems. We illustrate the benefits of ontological modelling and reasoning for service-oriented software architectures within the context of the Web Services.

Keywords: Model-Driven Architecture, Service-Oriented Architecture, Web Services, Modelling, Ontologies.

1 Introduction

Modelling has been recognised as an important aspect in the development of software architectures. Model-driven architecture (MDA) – a development framework proposed by the Object Management Group (OMG) – reflects this view [1]. MDA supports the development of component- and service-based software systems through modelling techniques based on notations such as UML. In particular service-oriented architecture (SOA) with its focus on the distributed development and deployment based on Internet and Web technologies requires an explicit representation of models [2,3]. Global software development (GSD) is another approach gaining importance recently that requires explicit, sharable models and descriptions in order to facilitate collaboration between developers on an abstract level as well as services on an implementation platform. Service-based platforms combined with the wide acceptance of Web technologies provide here suitable support. These contexts also necessitate a higher degree of reliability and dependability, which requires more rigour in the development activities. Formal reasoning is often required to automate development processes.

We will look here at the Web Services Framework (WSF) in particular as our platform [4,5]. The WSF is a service-based platform based on Internet- and Web-specific description languages, protocols, and core services. Modelling and describing services is essential for both providers and clients of services due to the distributed, inter-organisational nature of service-based development and deployment [6,7,8]. Sharing and reuse of models is a prerequisite for globally distributed software development.

Ontologies and ontology-based modelling have been proposed to enhance classical UML-based modelling. The essential benefits of ontologies are, firstly, interoperability and sharing and, secondly, advanced reasoning. Ontologies are sharable, extensible knowledge representation frameworks. They can also provide a further improvement on reasoning capabilities available in UML-extensions such as the Object Constraint Language (OCL) [9]. They can capture a wider range of functional and non-functional properties. Ontologies are consequently an ideal technology to support modelling for SOA and GSD. The potential of ontologies has already been recognised in MDA. The OMG has started the development of an Ontology Definition Metamodel (ODM) that can support ontological modelling for the MDA model layers [10]. The combination of ontologies and MDA has also been investigated in academic research, where the Web Ontology Language (OWL) has been integrated into MDA [11,12].

We propose an extension to these approaches. A layered, ontology-based modelling framework for software services can address the requirements of service-based and globally distributed software development. Similar to UML, which consists of different diagrammatic notations that address different modelling aspects, our framework will combine diferent ontology-based modelling techniques. The three different layers of the MDA framework – computation-independent, platform-independent, and platform-specific – shall be supported by three specific ontology techniques addressing the central concerns of these layers. These concerns are computation-independent domain modelling, platform-independent architectural configuration, and platform-specific service modelling. Architectural configuration is a central activity in software architecture. Service configurations and processes shall play a central role in our architecture framework.

We start with a short introduction to MDA, Web services, and ontologies in Section 2. In Section 3, we outline our ontology-based MDA framework for Web services. In the subsequent sections we address each layer separately – domain modelling in Section 4, architectural configuration in Section 5, and service implementation in Section 6. We end with a discussion of related work and some conclusions. We will use a Web-based e-learning system that we have developed and used over a couple of years as our case study. This system is currently being re-engineered based on a Web service-based architecture.

2 Services and Ontologies

Our objective is to open MDA to the Web Services Framework as the platform. Ontologies and Semantic Web technologies shall serve as the modelling approach.

2.1 Model-Driven Architecture

Model-driven architecture (MDA) is a software architecture framework that emphasises modelling as a central task in the development process of software systems [1]. MDA distinguishes three model layers:

– The computation-independent model layer (CIM) focusses on computation-independent aspects, i.e. modelling the domain-specific context of a system.
– The platform-independent model layer (PIM) aims to define a system in terms of computational abstractions. Typically, a computational paradigm or an abstract machine can form the basis of this modelling approach.
– The platform-specific model layer (PSM) consists of a platform model addressing the concepts and core services of the platform and an implementation-specific model addressing the concrete service architecture implementation.

MDA is typically applied useing UML for platform-independent modelling and considers CORBA as the platform with IDL for the platform-specific description.

2.2 Web Services

The Web Services Architecture WSA defines a Web service as a software system identified by a URI, whose public interfaces are defined and described using XML. Other systems can interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols. MDA is targeted towards service-based software systems, but the Web services platform was originally not considered. We will make the Web services platform our focus here, with service-oriented architecture (SOA) as the generic platform, the Web Services Framework as the technology-specific platform, and vendor-specific technologies such as service engines for SOAP-based service invocation forming the concrete platform infrastructure [5].

While first-generation Web service technology focussed on the use of services 'as-is' in single request-response interactions, the next generation of the Web services platform is more development-oriented [13]. The composition of services to processes is a major concern in current Web service research [14,15,16,17]. Service description and service discovery in repositories are essential elements of service development. These recent developments in the context of Web services have strengthened the importance of architectural questions. Behaviour and interaction processes are central modelling concerns for service-based software architectures. Ontology-based MDA can, as we will see, provide an ideal framework for this type of development support.

2.3 Ontologies

The Semantic Web is an initiative that aims to bring meaning to the Web [18,19]. Ontologies plays the central role in this endeavour. Ontologies are knowledge representation frameworks that allow knowledge to be shared. They combine terminological aspects with a formal logic. Ontologies usually consist of hierarchical definitions of important concepts of a domain and the description of properties of these concepts in terms of other concepts. An ontology is a model of a domain made available through the vocabulary of concepts and relationships.

Ontologies have already been used to support software engineering activities [20]. They have been exploited to support the annotation of Web services within the context of semantic Web services [21]. Ontologies are used to capture a variety of functional and non-functional properties of services (the terminological aspect of ontologies) and to retrieve matching provided services from repositories based on a client's requirements specification (the logical aspect of ontologies).

Ontology languages such as OWL are defined based on description logic, which allows the integration of formal reasoning with ontology-based modelling [22]. Description logics are particularly interesting for the software development context due to a correspondence between description logic and modal logics. Modal logics such as dynamic and temporal logics have been used extensively in the behavioural specification of software systems [23]. Dynamic logic forms a framework that captures the pre- and postcondition technique used in design-by-contract approaches [24] and specification languages such as OCL [9].

3 A Layered Ontological Modelling Framework

3.1 Modelling

MDA supports the architectural design of software systems. It integrates domain engineering with software architecture. MDA proposes a three-layered modelling framework addressing computation-independent, platform-independent, and platform-specific aspects. In our Web Services context, we have identified three central concerns that we can map to the MDA layers:

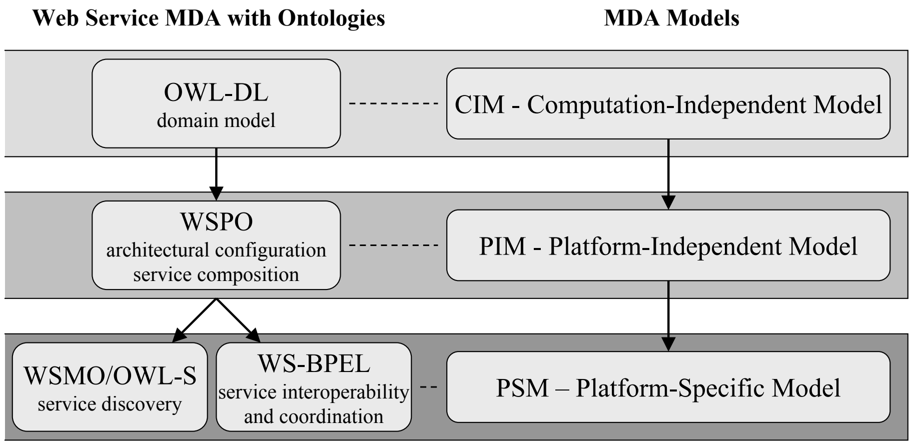

Web Service MDA with Ontologies
MDA Models
OWL-DL
domain model
CIM - Computation-Independent Model
WSPO
architectural configuration
service composition
PIM - Platform-Independent Model
WSMO/OWL-S
service discovery
WS-BPEL
service interoperability
and coordination
PSM – Platform-Specific Model

Fig. 1. Overview of the Ontology-based MDA Framework for Web Services

– Domain modelling is a concern that is independent of a concrete computational paradigm. Capturing the domain context of a service is essential for SOA as providers need to document the features of a service.
– Architectural configuration on a platform-independent level is important since service integration and composition are the central SOA activities.
– Modelling services within the given platform technology is important since service models have to be provided for discovery and service deployment in architectures and processes.

Our modelling framework – see Fig. 1 – consists of ontologies for all three layers. We will demonstrate that ontologies can address the concerns that have led to the definition of different model layers. For the Web services platform, an explicit sharable representation of these models for all layers is a requirement.

3.2 Logic and Semantics

Ontologies can address a variety of problems ranging from domain modelling to architectural configuration and semantic service description. The richness of an ontology language such as OWL-DL allows these different ontologies for different purposes to be developed [18].

In addition to providing notational modelling solutions for each of the concerns through different ontologies, ontology technology provides also the added benefit of enabling a uniform semantic framework for all three layers. Description logic is the formal foundation of many ontology languages, including the Web Ontology Language OWL.

The current work towards an Ontology Definition Metamodel (ODM), that has already been started by the OMG, will, once finished, provide an MOF-based semantic framework. We will discuss this aspect later on.

4 Computation-Independent Domain Modelling

4.1 Modelling Concern

The focus of the computation-independent modelling layer is the capture of domain properties. Here, often two viewpoints are distinguished. The information viewpoint captures structural aspects of information in form of concept hierarchies. The enterprise viewpoint looks at the behaviour and processes in a system. We add a third viewpoint addressing the structural aspects through compositional relationships.

4.2 Ontological Modelling

Ontologies consist of two basic entities – concepts of a domain and relationships between these concepts that express properties of one concept in terms of another concept. Classical ontologies relate concepts in a subclass hierarchy, which creates a taxonomy for a particular domain.

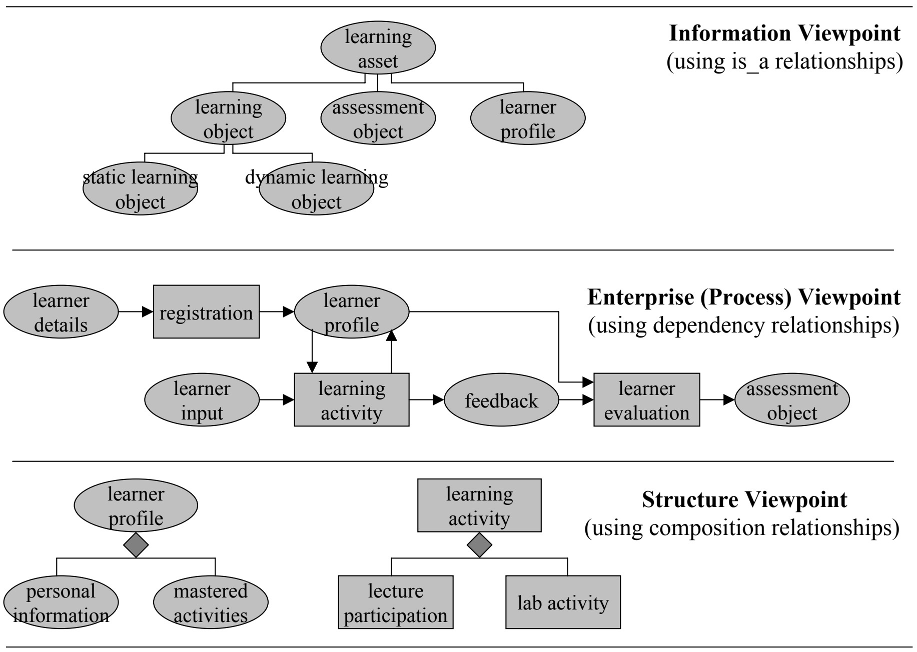


Fig. 2. CIM-level Excerpts from an E-Learning Domain Ontology

A single ontological notation, for example based on the Web Ontology Language OWL, can support the three viewpoints of the CIM layer.

- Two kinds of concepts – objects representing static information and processes presenting dynamic behaviour – can be distinguished in an ontology.
- The set of relationships shall comprise a subclass relationship for concept hierarchies (information viewpoint), a dependency relationship (enterprise viewpoint), and a component relationship (the structure viewpoint for both objects and processes).

The choice of relationship types here is critical to address the needs of process-centric domain modelling. Although, the concern here is domain modelling, domain activities and processes are central as they often form the starting point for further detailled models. Dependency relationships express how information objects are processed by process entities. Composition is important for both objects and processes.

We have illustrated computation-independent modelling in Fig. 2. In addition to the classical information viewpoint based on classification hierarchies and the enterprise viewpoint focusing on processes, we have also included a structural viewpoint addressing the compositional structure of objects and processes. Objects are elliptic entities such as learning object or assessment object. Processes are rectangular entities such as learning activity or evaluation. These entities – concepts in an ontology – are represented from the three viewpoints.

Other properties, such as sequencing constraints on processes can also be expressed in addition to concepts and relationships. Iteration, choice, concurrency, or sequence are process combinators that are often better expressed in a separate sublanguage. For instance, individual activity steps of a learning activity could be sequenced using additional constraints.

4.3 Ontological Reasoning

The reasoning capabilities of an ontological framework can be utilised in different ways for this form of domain modelling:

– The internal consistency of a model can be checked. For instance, cyclical definitions in concept taxonomies can be recognised.
– Inference rules can also be used to query an ontology. For instance, rules about transitional process behaviour (based on dependency relationships) can be used to determine the input/output behaviour of composite processes.

The description logic on which an ontology language like OWL-DL is based provides the formal framework here [22].

In the context of the e-learning example, an inference engine can be used to compile all prerequisites of a sequence of learning activities. It could also be used to check whether the learning outcomes of the first activity in a sequence satisfy the prerequisites of the next activity.

5 Platform-Independent Architecture Modelling

5.1 Modelling Concern

Platform-independent modelling introduces a computational paradigm into the modelling process. Service-oriented architecture is this paradigm for the Web Services Framework. In the context of service-based software, the architectural design of a software system is of central importance. Behaviour and service processes are part of the architectural configuration of a system [15]. Architectural configuration addresses the interaction processes (remote invocation and service activation) between different services in a software system.

5.2 Ontological Modelling

Various service ontologies exist [25]. WSPO – the Web Service Process Ontology – can be distinguished from other service ontologies such as OWL-S [21] and WSMO [26] through its process-orientation [27,28,29]. In WSPO, the focus is on the behaviour of software systems. Relationships of the ontology are interpreted as accessibility relations between system states. This is in fact an encoding of a (modal) dynamic logic in a description logic format [30]. WSPO ontologies are based on a common template, see Fig. 3:

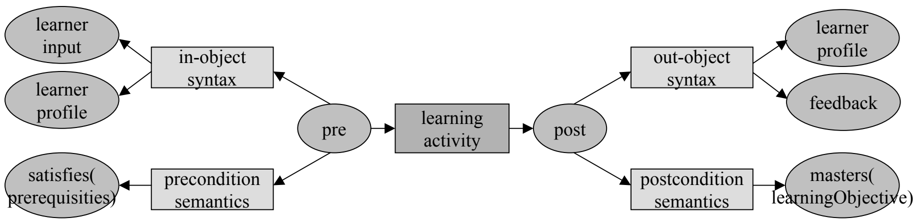


Fig. 3. Ontological Service Process Template (WSPO) – applied to E-Learning

– The central concepts are the system states – pre- and poststates of state transition-based services. Other concepts capture service parameters (in- and out parameters) and conditions (such as pre- and postconditions).
– Two forms of relationships characterise this ontology. The central relationship type is the service or service process itself. These so-called transitional relationships are enhanced by a process combinator sublanguage comprising the operators sequence, choice, iteration, and parallel composition. This relationship sublanguage allows process expressions to be formulated. Auxiliary relationship types are so-called descriptional relationships, which associate the auxiliary concepts to the states.

The architecture- and process-oriented PIM model of the e-learning example focuses on the activities and how they are combined to processes, see Fig. 3. The process combinators that we used to model the e-learning activity service are ';' (sequential composition), '!' (iteration), '+' (choice), and '||' (parallel composition). The operators are interpreted by their usual set-theoretic semantics, e.g. iteration is defined as the transitive closure of the relation that interprets the argument and the non-deterministic choice is interpreted by the union of both argument interpretations. The symbol $\circ$ is used to denote the application of a process to a given list of parameters. The process

$$lecture;\ !(\ labExercise1 + labExercise2\ );\ \ selfAssessment$$

describes a sequence of lecture participation, an iteration of a choice of two lab exercises, and a final self-assessment. This can be represented in WSPO as a composed relationship expression:

$$\begin{array}{l} lecture \circ profile; \\ !\ (\ labExercise1 \circ (profile, input1);\ labExercise2 \circ (profile, input2)\ ); \\ selfAssessment \circ profile \end{array}$$

While architecture is the focus of this model layer, the approach we discussed does not qualify as an architecture description language (ADL) [31], although the aim is also the separation of computation (within services) and communication (interaction processes between services). ADLs usually provide notational means to describe components (here services), connectors (channels between services), and configurations (the assembly of instantiations of components and connectors). Our approach comes close to this aim by allowing services as components and process expressions as configurations to be represented.

5.3 Ontological Reasoning

The close link to modal logic allows modal reasoning about reactive systems to be incorporated.

- Dynamic logic, for instance, incorporates pre- and postcondition-based reasoning. Matching between client requirements and service properties in terms of abstract functional behaviour can be decided using this technique. This link also allows the integration of a refinement technique.
- The process expression language in WSPO is enhanced by supporting behavioural theory from temporal logics and process calculi. A notion of simulation allows process expressions to be compared. This can be used in matching.

Matching is a common problem that needs to be addressed if a new component, such as a service or process here, has to be embedded into a given context. For any given state, the process developer might require

$$\begin{array}{l} \forall preCond\ .\ (profile.masteredActivities \in activity.prerequisite) \\ \forall learning\ activity\ .\ \forall postCond\ . \\ \qquad\qquad (activity.objective \in profile.masteredActvities) \end{array}$$

which would be satisfied by a provided service

$$\begin{array}{l} \forall preCond\ .\ true \\ \forall learning\ activity\ .\ \forall postCond\ . \\ \qquad\qquad (activity.objective \in profile.masteredActvities) \wedge \\ \qquad\qquad (typeof(lastActivity) =\ 'labExercise') \end{array}$$

based on a refinement condition (weakening the precondition and strengthening the postcondition). The dot-notation used in the conditions refers to a component of the object. Quantified expressions are used to express these safety conditions. The postcondition in this example states that by carrying out the activity, the intended activity objective is accomplished and can therefore be added to the mastered activities of the learner in her/his profile.

5.4 Transformation

Transformations between the layers are crucial. A high degree of tool support and automation is necessary for an MDA framework in general. Although a detailed discussion of transformations in our framework is beyond the scope of this paper, we will outline the principles here. We have investigated transformations for this framework in more detail in [29].

The CIM-to-PIM transformation involves the mapping of a domain ontology into a process-centric service ontology. The following steps need to be considered:

– Service identification. In our case, the domain model is already service-oriented and services are clearly marked.
– Model mapping. We have defined a WSPO ontology template applicable to a single or a composed service. For the composed service processes, the actual process description comes from a separate constraint. The instantiation of this template leads to a service-specific ontology.

6 Platform-Specific Service Modelling

6.1 Modelling Concern

Platform-specific modelling (PSM) relates the previous layers to the concrete constraints of the chosen platform. It usually consists of two models – the platform model that describes the specifics of the platform and the implementation specific model that captures the essentials of the implementation languages.

6.2 Ontological Modelling

The platform here is the Web Services platform, on which a number of different languages are used. We will look at two of these languages here:

– Development support: We have chosen WSMO as one of the widely discussed and used service ontologies that aims at describing a service on an abstract level. WSMO incorporates some of the functional behaviour specification of WSPO, but also provides support for a wide range of non-functional properties, see Fig. 4. The aim of WSMO is to provide an interoperable form for the semantic description of services to support their discovery in repositories.
– Deployment support: Another aspect of service-oriented architecture is service composition – often the term service collaboration is used to indicate the distributed nature of service architectures. WS-BPEL is a business process description language that supports Web service orchestration (collaboration described from a local services' point of view). WSPO already captures the essentials of service and process interaction. This can easily be translated into WS-BPEL specifications.

Both are service-centric implementation languages. Interoperability is a central issue in both cases.

The learning activity service that we have focussed on in our case study could both be published (using WSMO) and integrated in a service process orchestration (using WS-BPEL):

– WSMO descriptions capture syntactical and semantic service descriptions. In this way it is similar to WSPO. It adds, however, various non-functional aspects that can be included into the discovery and matching task. We have added two non-functional properties to the learning activity descriptions – the location as an interface-related aspect and the security infrastructure as a capability issue, see Fig. 4.

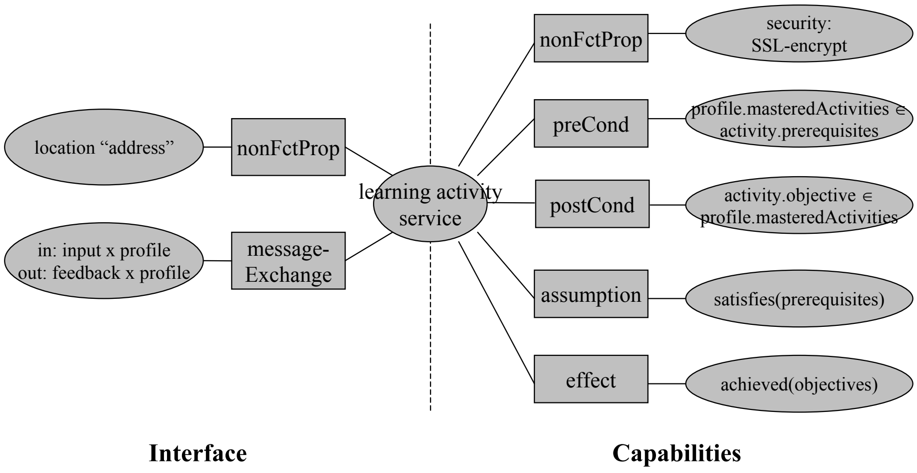


Fig. 4. Ontological Service Template (WSMO) – applied to an E-Learning Context

Standardised description and invocation formats enable interoperability. A key objective is the provision of services. For instance, the learning activity could be advertised as a reusable content service in a learning technology repository. Required functionality can be retrieved from other locations. An example are registration and authentication features.

– WS-BPEL is an XML-based notation, based on an operator calculus similar to ours. Based on simple actions <`action`>, which describe simple request or response interactions, combinators such as <`sequence`> or <`flow`> can be used to define orchestrated service processes. Orchestration is an internal perspective on process assembly and interaction.

6.3 Ontological Reasoning

Again, a formal ontology basis enables further forms of reasoning. For the WSMO context, the matching notion can be extended to comprise a variety of functional and non-functional properties. The main aim of OWL-S and WSMO is the improved support of semantic Web service description and discovery compared to syntactical formats such as the Web Services Description Language (WSDL).

For WS-BPEL, classical process calculus-based analyses, e.g. a deadlock analysis, can be addressed. Although not part of WS-BPEL itself, a formulation of WS-BPEL using a process calculus would enable these analyses.

6.4 Transformation

The PIM-to-PSM transformation encompasses two mappings for the two different platform languages we support:

– The WSPO-to-WSMO mapping extracts information specific to individual services from a WSPO model. This comprises the syntactical elements (in- and out-objects) and the semantic information (pre- and postconditions).
– The WSPO-to-WS-BPEL extracts the process definitions and converts them into BPEL process expressions. It also uses the syntactical information to define the individual services.

Both mappings can create skeletons with partial information that would need to be completed by a software developer. While transformations are essential, we will not discuss them here – see [29] for a more detailed investigation.

7 Related Work

WSMO [26] and OWL-S [21] are the two predominant examples of service ontologies. Service ontologies are ontologies to describe Web services, aiming to support their semantics-based discovery in Web service registries. WSMO is not an ontology, as OWL-S is, but rather a framework in which ontologies can be created. We have used WSMO here to illustrate issues for a Web service platform-specific modelling approach. The Web Service Process Ontology WSPO [27,28], which we have used for platform-independent modelling, is also a service ontology, but its focus is the support of description and reasoning about service composition and service-based architectural configuration. Both OWL-S and WSPO are or can be written in OWL-DL. WSMO is similar to our endeavour here, since it is a framework of what can be seen as layered ontology descriptions. We have introduced technical aspects of WSMO descriptions in Section 6. WSMO supports the description of services in terms more abstract assumptions and goals and more concrete pre- and postconditions.

Some developments have started exploiting the connection between OWL and MDA. In [32], OWL and MDA are integrated, i.e. an MDA-based ontology architecture is defined. This architecture includes aspects of an ontology metamodel and a UML profile for ontologies. A transformation of the UML ontology to OWL is implemented. The work by [11,32] and the OMG [1,10], however, needs to be carried further to address the ontology-based modelling and reasoning of service-based architectures. In particular, the Web Services Framework needs to be addressed in the context of Web-based ontology technology.

Our framework has to be looked at in the context of the MDA initiatives by the OMG. The OMG supports selected modelling notations and platforms through an adoption process. Examples of OMG-adopted techniques are UML as the modelling notation and CORBA as the platform [1]. While Web technologies have not adopted so far, the need for a specific MDA solution for the Web context is a primary concern. The ubiquity of the Web and the existence of standardised and accepted platform and modelling technology justify this requirement. The current OMG initiative to define and standardise an ontology metamodel (ODM) will allow the integration of our framework with OMG standards [10]. ODM will provide mappings to OWL-DL and also a UML2 profile for ontologies to make the graphical UML notation available. This might lead to a 'Unified Ontology Language' in the future, i.e. OWL in a UML-style notation [12]. A UML2 profile is about the use of the UML notation, which would allow ontologies to be transformed into UML notation. MOF2 compliancy for ODM is requested to facilitate tool support. XMI, i.e. production rules using XSLT, can be used to export model representations to XML, e.g. to generate XML Schemas from models using the production rules. The ontology definition metamodel (ODM) would allow an integration with UML-style modelling due to its support of OWL. ODM, however, is a standard addressing ontology description, but not reasoning. The reasoning component, which is important in our framework, would need to be addressed in addition to the standard.

8 Conclusions

Ontology technology offers a range of benefits for modelling activities in the MDA context.

- The formal definition based on description logics allows extensive reasoning to be used.
- Ontologies are sharable knowledge representation formats. Ontologies can easily be modified and extended.

Ontologies combine a terminological framework with a logical framework. It is this combination that we have used to enhance classical modelling techniques.

The benefits match in particular the requirements of a platform such as the Web Services Framework, where often globally distributed software development is the main style of development that relies on interoperable data formats and dependable service architectures. In heterogeneous environments and cross-organisational development, information about a variety of service aspects – as it can be represented in ontologies – is vital.

MDA defines a development process, addressing different concerns at each stage. We have identified process-oriented domain modelling, architectural configuration, and service implementation modelling as the three central concerns for the development with the Web Service Framework as the platform. We have demonstrated that ontology technology can provide an integrated, coherent solution for these concerns at all three modelling layers.

CORBA and UML are OMG-adopted technologies. The adoption process provides OMG support for a particular technology, either a platform or language. Web technologies have not been adopted so far. However, the ubiquity of the Web will require a solution in the future. The current OMG attempt to define an ontology definition metamodel ODM that includes mappings to OWL-DL and also a UML profile for ontologies is a first step integrating OMG with Web technologies.

We have neglected one central problem of a Web ontology-based MDA approach. Transformations between the individual layers need to be defined and, to a high degree, automated in order to make MDA feasible. The definition of a transformation framework is beyond the scope of this paper. Two transformation steps have to be addressed. Both are transformations between different ontologies. We have devised a graph-based transformation centred around the service and process elements; see [29] for more details.

References

1. Object Management Group. *MDA Guide V1.0.1.* OMG, 2003.
2. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services – Concepts, Architectures and Applications.* Springer-Verlag, 2004.
3. E. Newcomer and G. Lomow. *Understanding SOA with Web Services.* Addison-Wesley, 2005.
4. World Wide Web Consortium. *Web Services Framework.* `http://www.w3.org/2002/ws, 2004.` (visited 08/04/2005).
5. World Wide Web Consortium. *Web Services Architecture Definition Document.* http://www.w3.org/2002/ws/arch, 2003.
6. The WS-BPEL Coalition. *WS-BPEL Business Process Execution Language for Web Services – Specification Version 1.1.* `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel, 2004.` (visited 08/04/2005).
7. C. Peltz. Web Service orchestration and choreography: a look at WSCI and BPEL4WS. *Web Services Journal*, 3(7), 2003.
8. D.J. Mandell and S.A. McIllraith. Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. In D. Fensel, K.P. Sycara, and J. Mylopoulos, editors, *Proc. International Semantic Web Conference ISWC'2003*, pages 227–226. Springer-Verlag, LNCS 2870, 2003.
9. J.B. Warmer and A.G. Kleppe. *The Object Constraint Language – Precise Modeling With UML.* Addison-Wesley, 1998.
10. Object Management Group. *Ontology Definition Metamodel - Request For Proposal (OMG Document: as/2003-03-40).* OMG, 2003.
11. D. Gašević, V. Devedžić, and D. Djurić. MDA Standards for Ontology Development – Tutorial. In *International Conference on Web Engineering ICWE2004*, 2004.
12. D. Gašević, V. Devedžić, and V. Damjanović. Analysis of MDA Support for Ontological Engineering. In *Proceedings of the 4th International Workshop on Computational Intelligence and Information Technologies*, pages 55–58, 2003.
13. J. Williams and J. Baty. Building a Loosely Coupled Infrastructure for Web Services. In *Proc. International Conference on Web Services ICWS'2003.* 2003.
14. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transacions on Software Engineering and Methodology*, 6(3):213–249, 1997.
15. F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. *ACM Transactions on Software Engineering*, 28(11):1056–1075, 2002.
16. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice (2nd Edition).* SEI Series in Software Engineering. Addison-Wesley, 2003.
17. N. Desai and M. Singh. Protocol-Based Business Process Modeling and Enactment. In *International Conference on Web Services ICWS 2004*, pages 124–133. IEEE Press, 2004.
18. W3C Semantic Web Activity. Semantic Web Activity Statement, 2004. http://www.w3.org/2001/sw. (visited 06/12/2004).
19. M.C. Daconta, L.J. Obrst, and K.T. Klein. *The Semantic Web.* Wiley, 2003.

20. M. Paolucci, T. Kawamura, T.R. Payne, and K. Sycara. Semantic Matching of Web Services Capabilities. In I. Horrocks and J. Hendler, editors, *Proc. First International Semantic Web Conference ISWC 2002*, LNCS 2342, pages 279–291. Springer-Verlag, 2002.
21. DAML-S Coalition. DAML-S: Web Services Description for the Semantic Web. In I. Horrocks and J. Hendler, editors, *Proc. First International Semantic Web Conference ISWC 2002*, LNCS 2342, pages 279–291. Springer-Verlag, 2002.
22. F. Baader, D. McGuiness, D. Nardi, and P.P. Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.
23. D. Kozen and J. Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 789–840. Elsevier, 1990.
24. Bertrand Meyer. Applying Design by Contract. *Computer*, pages 40–51, October 1992.
25. T. Payne and O. Lassila. Semantic Web Services. *IEEE Intelligent Systems*, 19(4), 2004.
26. R. Lara, D. Roman, A. Polleres, and D. Fensel. A Conceptual Comparison of WSMO and OWL-S. In L.-J. Zhang and M. Jeckle, editors, *European Conference on Web Services ECOWS 2004*, pages 254–269. Springer-Verlag. LNCS 3250, 2004.
27. C. Pahl. An Ontology for Software Component Matching. In M. Pezzè, editor, *Proc. Fundamental Approaches to Software Engineering FASE'2003*, pages 6–21. Springer-Verlag, LNCS 2621, 2003.
28. C. Pahl and M. Casey. Ontology Support for Web Service Processes. In *Proc. European Software Engineering Conference and Foundations of Software Engineering ESEC/FSE'03*. ACM Press, 2003.
29. C. Pahl. Ontology Transformation and Reasoning for Model-Driven Architecture. In *International Conference on Ontologies, Databases and Applications of Semantics ODBASE'05*. Springer LNCS Series, 2005. (submitted).
30. K. Schild. A Correspondence Theory for Terminological Logics: Preliminary Report. In *Proc. 12th Int. Joint Conference on Artificial Intelligence, Sydney, Australia*. 1991.
31. N. Medvidovic and R.N. Taylor. A Classification and Comparison framework for Software Architecture Description Languages. In *Proceedings European Conference on Software Engineering / International Symposium on Foundations of Software Engineering ESEC/FSE'97*, pages 60–76. Springer-Verlag, 1997.
32. D. Djurić. MDA-based Ontology Infrastructure. *Computer Science and Information Systems (ComSIS)*, 1(1):91–116, 2004.

Model-Driven Development – Hot Spots in Business Information Systems

Bernhard Humm[1], Ulf Schreier[2], and Johannes Siedersleben[1]

[1] sd&m Research, Munich, Germany
{Bernhard.Humm, Johannes.Siedersleben}@sdm.de
[2] University of Applied Sciences Furtwangen, Furtwangen, Germany
Ulf.Schreier@fh-furtwangen.de

Abstract. Model-driven development (MDD) is an important technology since it helps to reduce the cost of software development and maintenance. Furthermore, it may increase the quality of resulting systems. However, it is naive to try to generate complete complex systems out of a single model. This paper presents hot spot domains of business information systems where MDD is particularly applicable. For those domains, proven techniques for combining MDD with software architecture principles are presented. This distils the experience with MDD at sd&m for more than fifteen years.

1 Introduction

Model-driven architecture (MDA) is currently a hype topic. There are tool vendors that promise the generation of complete systems by the press of a button and an increase of productivity by a factor up to 47 [CARE].

In his famous article "No silver bullet" [Bro86] nearly twenty years ago, Fred Brooks pinpointed automatic programming as one of the chimeras of software engineering. MDA, wrongly understood, is automatic programming in Brooks' sense. But is there a right understanding and does it really work in practice?

Before going on, we introduce the term *model-driven development (MDD)* which we prefer to the term MDA. The term MDA may be misleading in the sense that it is not the architecture of the resulting system but the architecture of the software development environment which is addressed. MDD is more than the MDA initiative of the Object Management Group [OMG].

MDD is not new and has proven its value, even though productivity increases of a factor of 47 [CARE] have never been reached. At sd&m – a German software company with more than 900 employees – MDD has been successfully applied to the domain of business information systems for more than fifteen years[1], initially with text-based models [Den93, Sch93, Kel94]. This paper condenses the essence of our learning in MDD from more than a hundred projects with a total volume of more than thousand person years.

[1] Numerous colleagues have been involved in research on MDA at sd&m, currently including Thomas George, Oliver Juwig, Martin Kempa, Vera Kießling, Jürgen Krey, Friedemann Ludwig, Marcello Mariucci, Zoltan Mann, Klaus Nünninghoff, Frank Salger, Thomas Tensi, Ludger Unland, and Angelika Wittek.

2 What is MDA?

MDA is a standardization initiative of the OMG in the area of MDD, started in 2001. The central standardization items are:

- (Meta) Modelling languages: The *Unified Modelling Language* [UML] and its various extensions (UML profile, Executable UML, OCL) as the preferred modelling language, *Meta Object Facility* [MOF] as the basis for meta modelling, and *XML Metadata Interchange* [XMI] as the standard for the interchange of models
- Model transformation: *Query, View, Transformation* [QVT] as the emerging standard for the transformation between object-oriented models.

The main principle of MDA is illustrated in Fig. 1.

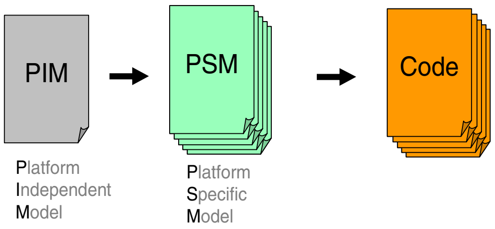


Fig. 1. Main generation steps of MDA

The code of a system is created by successive transformation and generation steps. The starting point is the *platform-independent model (PIM).* MDA does not specify precisely what a platform is. Hence, the independence from a platform is not specified either. A common understanding is that a PIM for business information systems specifies business application logic only. The *platform-specific model (PSM)* enhances the PIM with details of a technical execution platform, e.g., J2EE or .NET. PIM and PSM use UML diagrams and profiles as concrete syntax. Out of the PSM, code is generated in a particular programming language with text-oriented concrete syntax, e.g., in Java or C#. Conceptually, code in a programming language is a model, too, since all – PIM, PSM and code – are based on abstract syntax (see for instance [GS04]).

The transformation and generation steps successively expand the models by particular behaviour. The expansion logic lies in the transformers and generators themselves and may get additional input by marks specified in the input models. The transformation and generation steps are strictly unidirectional from PIM to code. Round-trip engineering is not part of the concept.

Naively understood, MDA may be tempting to be applied in the following way: the business application logic of a complete business information system or even a whole family of systems is specified completely in the PIM. I.e., the complete object model as well as all algorithmic logic is specified in an UML variant that is executable. All the rest is added automatically via transformers and generators and the resulting code is generated without the need of manual programming.

This is the naïve view that Brooks pinpointed as the chimera of automatic programming. It is obvious that if you try to express all business application logic in the model the modelling language must be as powerful as the programming language and the model is as voluminous as the application program. You abuse the model as a programming language which is not the appropriate level of abstraction for this domain. The 4GL tools of the 1990s have gone exactly this naïve way and have failed.

However, MDD is general enough to be useable and successful if applied in an intelligent way.

3 Separating Business Application Logic from Technology

What are the main goals of MDD? To reduce development and maintenance costs and to increase the quality of the resulting software systems. What are the means by which MDD aspires to reach the goals? Separating concerns, particularly separating business application logic from technology in different models.

There are alternatives and proven approaches that support the separation of business application logic from technology like, e.g., frameworks and component architectures. *Quasar* [Sie04, HS04, [Hum04], Sie03, DS00] stands for *quality software architecture*. It consists of a set of design principles, a reference architecture for business information systems and a set of technical components. One core principle of Quasar is the separation of business application logic from technology. We briefly introduce Quasar to explain the synergies between MDD and Quasar in the rest of the paper.

Fig. 2 gives an overview of the reference architecture for business information systems as a UML component diagram.

The reference architecture illustrates the separation of business application logic from technology *on the component level*. Business application logic is concentrated in application components (stereotype `<<A component>>` – “A” stands for “application”) in the application kernel and for dialogs. An application kernel facade separates the application components from platform-specifics. E.g., application logic may be implemented in pure Java; specifics of a J2EE application server are hidden in the application kernel facade. The application kernel facade also supports location transparency by providing means for client / server communications.

Technical services are encapsulated in abstract technical components (stereotype `<<Abstract T Component>>`). Abstract technical components are specified by their interfaces. In a real system, *abstract* components are implemented by *concrete* products like, e.g., TopLink [TopLink], Hibernate [Hibernate], or QuasarPersistence [Ern04, OpenQuasar] for object / relational mapping (`Persistence`). In contrast to a persistence layer as the artefact of a generator, such a persistence component is developed, tested, optimized, and deployed separately from the application. For further details of the reference architecture, see [HHS04].

It is relatively easy to draw a diagram that postulates the separation of business application logic and technology on the component level. However, it requires a lot of expertise to maintain the separation in the internals of all components. We have proven the feasibility of such a separation in numerous large-scale projects. Apparently, MDD concepts are extremely helpful in effectively implementing such a separation of concerns. In the following sections, we present details.

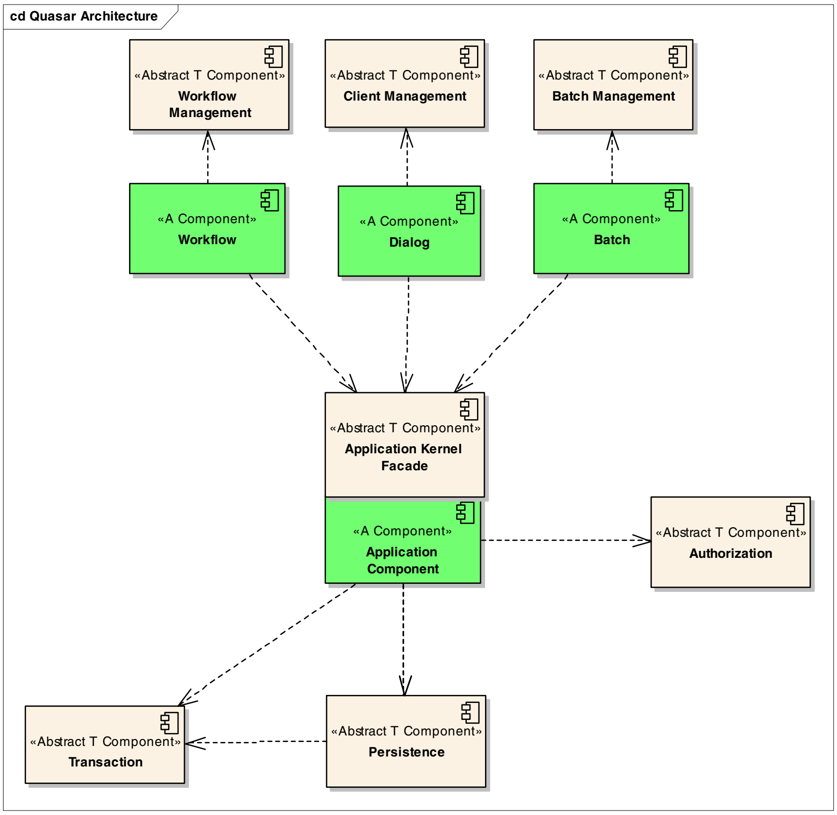

cd Quasar Architecture
«Abstract T Component» Workflow Management
«Abstract T Component» Client Management
«Abstract T Component» Batch Management
«A Component» Workflow
«A Component» Dialog
«A Component» Batch
«Abstract T Component» Application Kernel Facade
«A Component» Application Component
«Abstract T Component» Authorization
«Abstract T Component» Transaction
«Abstract T Component» Persistence

Fig. 2. Quasar reference architecture for business information systems

4 Model-Driven Components

One of the most important concepts for the architecture of large-scale systems like business information systems is *component-orientation*.(e.g., [Szy02, Sie04]) Components exhibit operative interfaces they export and import. Components use the services of other components via their operative interfaces. From the point of view of a using component, it is irrelevant whether a used component has been programmed manually, generated in parts or generated completely using MDD. At their operative interfaces, manually programmed components behave identically to generated ones. The dependence on models is the secret of a component, reflected by additional administrative interfaces. Components that depend on models at development-time or at run-time we call *model-driven components* (see also [Teu04]).

We have identified a number of *hotspot domains* in the context of business information systems that have proven particularly suitable for model driven components. In line with common terminology, we use the term *domains*, not to be confused with business application domains like, e.g., financial services.

We refer to custom software projects. For specific business application domains like, e.g., financial services, MDD can also be used for the development of software product families.

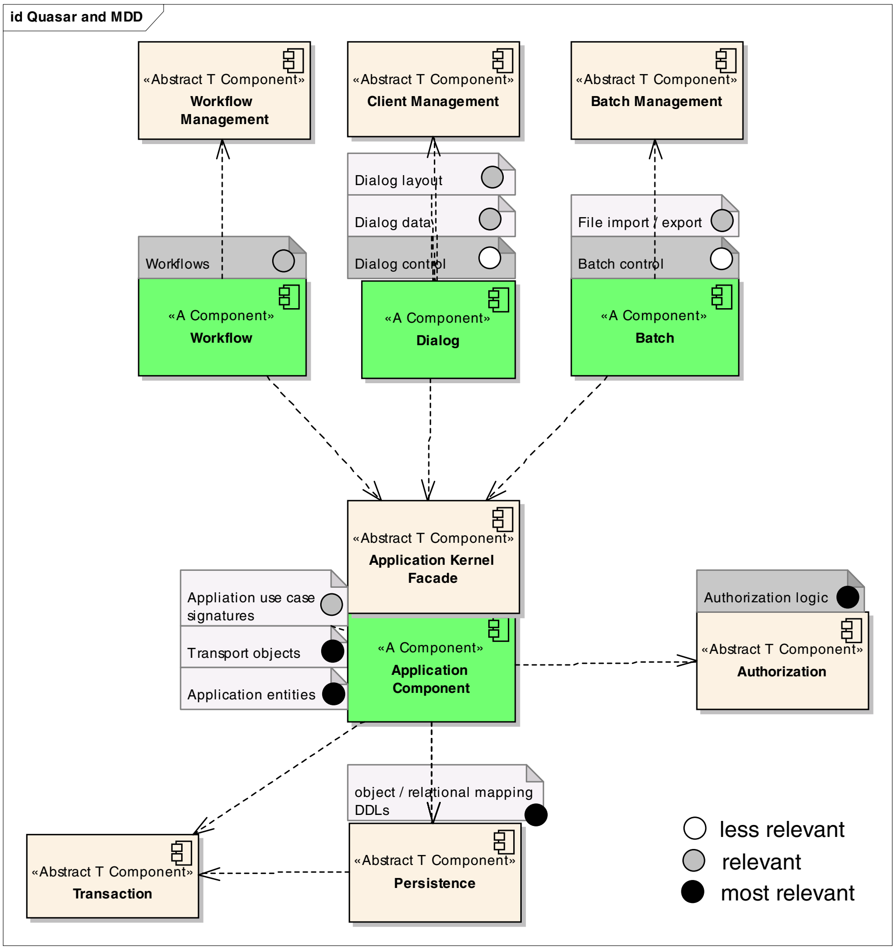

id Quasar and MDD
«Abstract T Component» Workflow Management
«Abstract T Component» Client Management
«Abstract T Component» Batch Management
Dialog layout
Dialog data
Dialog control
File import / export
Batch control
Workflows
«A Component» Workflow
«A Component» Dialog
«A Component» Batch
«Abstract T Component» Application Kernel Facade
Appliation use case signatures
Transport objects
Application entities
«A Component» Application Component
Authorization logic
«Abstract T Component» Authorization
object / relational mapping DDLs
«Abstract T Component» Transaction
«Abstract T Component» Persistence
less relevant
relevant
most relevant

Fig. 3. MDD domains for business information systems

The selection of hot spot domains best suited for MDD is based on extensive project experience at sd&m over the last fifteen years. Fig. 3 gives an overview in the context of the Quasar reference architecture for business information systems.

We have ranked these domains with respect to their relevance in those projects. The ranking is illustrated in Fig. 3 in form of bullets. *Most relevant* (black bullet) means that MDD has been used in nearly every project. *Less relevant* means that MDD has been used in practice, but only rarely. *Relevant* means that depending on the complexity of the problem, MDD or manual programming has been chosen: whereas MDD is more suitable for simple domains, the full power of programming languages is needed in complex domains.

The following *data domains* (light note symbols in Fig. 3) are particularly suitable for MDD.

- *Application entities*: Application entity classes are often generated out of data models and possibly manually extended by application logic. Meta data regarding application entities, e.g., descriptors for the application kernel facade or application servers can be generated, too.
- *Object / relational mapping and DDLs*: table definitions (DDL statements) may be generated out of the data model and possibly optimised, e.g., by denormalization. Then, also the object / relational mapping data for persistence components may be generated.
- *Transport objects*: Transport objects are simple data structures that are used to decouple clients and application components as well as application components from each other. They are used as parameter types of the exported interfaces instead of complex entity objects. Transport object classes as well as the conversion routines between entity objects and transport objects are often be generated from data models.
- *File import / export*: Often, application data are imported and exported from and to files in batch programs. From data models, conversion routines from files to entity objects and vice versa may be generated. For complex transformation, script languages are commonly used.
- *Dialog data*: Dialog data are architecturally decoupled from application component data, often also physically in a client / server architecture. However, their structure is often similar to the one of application entities. Then, dialog data may be generated from data models. If the structure is different manual programming is more appropriate.
- *Dialog layout*: Stereotype user interfaces may be modelled in a specific layout modelling language. The code for the dialog layout may then be generated from these models. For dynamic user interface layouts, manual programming is more suitable.
- *Application use case signatures*: Use case classes form the interfaces of application components. From service specifications, the signatures of use case classes may be generated. Meta data regarding application use cases, e.g., descriptors for the application kernel facade or application servers can be generated, too.

The following *behavioural domains* (dark note symbols in Fig. 3) are particularly suited for MDD.

- *Workflows*: model data for workflow or use case management systems may be generated from process models. Highly complex workflows need manual programming.
- *Batch control*: Job control may be generated from process models. In other cases, script languages are used.
- *Authorization logic*: Authorization control is usually based on the combination of role and user group models on the one side and access right models on the other side. Those models are typically interpreted by authorization components.
- *Dialog control*: Dialog control logic may be specified with state machines. Respective code for a dialog controller may be generated. However, dialog control is often more conveniently expressed by a few lines of code.

The selection shows that the domain of complex business application logic is not suited for MDD. Such logic is best programmed manually: a modern programming language is the most appropriate language for this domain.

This leads us to a key learning: identify domains where MDD is appropriate and use *domain-specific modelling languages*. Use programming languages where appropriate. Do not try to generate a complete system from a single model. However, there are a number of domains where MDD can substantially reduce tedious programming work.

The idea of domain-specific modelling languages is also central to Microsoft's approach to MDD, called software factory [GS04]. OMG's MDA also supports domain-specific modelling languages with the help of UML profiles.

5 Generation Versus Interpretation

Apart from generating code from models, models may be *interpreted* at run-time. Generation and interpretation are conceptually similar and both variants of MDD. Theoretically, generation can always be replaced by interpretation and vice versa. In practice, some domains are more suited for generation and others for interpretation. Generated code is more convenient to debug, is more robust due to compile-time checks and usually exhibits better performance. Interpretation allows modifications without recompilation and is more responsive to run-time information, e.g., current user information.

Interpretation is usually preferable in the behavioural domains whereas generation is usually preferable in the data domains. This is the case since behavioural domains are often responsive to run-time information. Since in data domains, this is rarely the case, the more robust generator approach is usually chosen. It is the decision of the chief designer to decide between generative and interpretative approaches based on project specifics.

According to the MDA Guide [MM03], the OMG also encompasses generation as well as interpretation in MDA.

6 Integrating MDD Code with Manually Programmed Code

6.1 Separating Business Application Code from Technical Code

Accepting the fact that MDD is appropriate for specific domains only, one has to live with a coexistence of generated (respectively interpreted) and manually programmed code. MDA tools usually offer the mechanism of *protected areas* to separate generated code from manually programmed one. Naively, one could use protected areas to separate platform-specific, generated code from platform-independent, manually programmed code. This naive use leads to code in which business application code is mixed with technical code.

We propose an alternative to the naive approach that allows real separation of concerns on the code level without losing the advantages of MDD. See Fig. 4.

On the code level, we separate clearly between business application code (*A software*) and technical code (*T software*) [Sie04]. The PIM may be used to generate frames for the implementation of the business application logic, e.g., entity classes. The application logic proper is implemented manually. Technical components like,

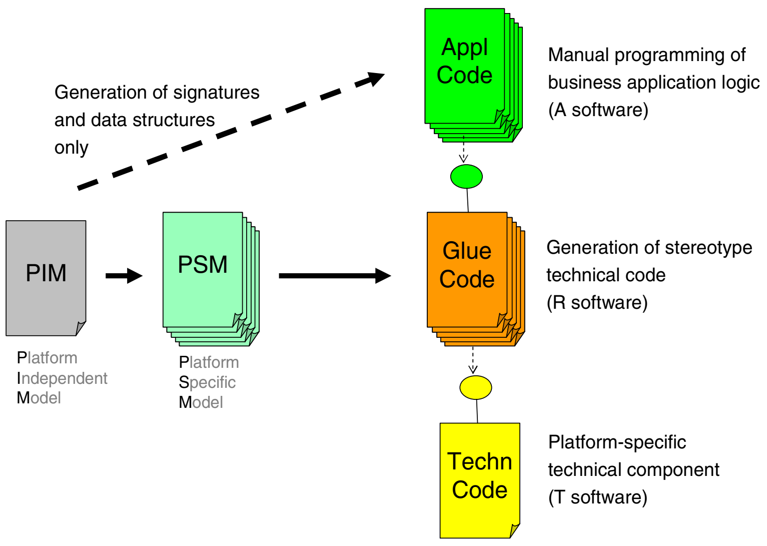

Generation of signatures and data structures only
Appl Code
Manual programming of business application logic (A software)
PIM
Platform Independent Model
PSM
Platform Specific Model
Glue Code
Generation of stereotype technical code (R software)
Techn Code
Platform-specific technical component (T software)

Fig. 4. Separating business application code from technical code using MDD

e.g., TopLink, Hibernate, or QuasarPersistence for object / relational mapping (persistence), are installed and integrated. The glue code (*R software*) is stereotype technical code which translates between A software and T software. It is generated out of the PSM. Alternatively, a PIM enhanced by marks may substitute an explicit PSM.

6.2 Example

We explain the principle by the example of application entity classes. See Fig. 5.

The example shows an entity `Customer` of a business information system with simple attributes `name`, `address` and a complex business application method `checkCreditRating()`.

In the naive solution, a Class Customer is generated which mixes specifics of the persistence product used and manually programmed business application logic. We present four exemplary alternatives:

1. Separation via controllers
2. Separation via inheritance
3. Separation via delegation
4. Separation via annotation

6.3 Alternative 1: Separation Via Controllers

In Alternative 1, entity objects (`CustomerGen`) consist of R software only. They are generated, contain the attributes of the customer entity and specifics of the technical persistence component (e.g., inheritance from a `Persistent` base class). However, they contain no application logic whatsoever. For each entity of the PIM, a controller class (`CustomerController`) is generated as a singleton [GHJ+95]. All business application logic is implemented there.

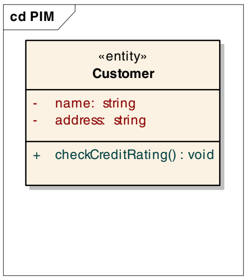


Fig. 5. Entity example

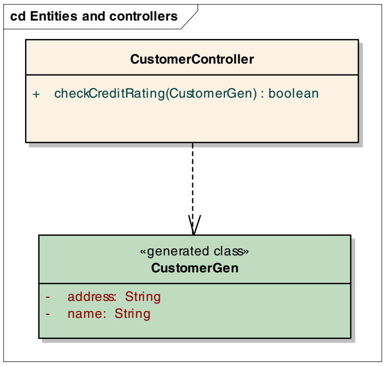


Fig. 6. Separation via controllers

Alternative 1 is the simplest approach. However, it is the least object-oriented one since it separates data from behaviour in different classes.

6.4 Alternative 2: Separation Via Inheritance

In the second alternative, a class `CustomerGen` is generated which contains the attributes and specifics of the persistence component (R code). A subclass `Customer` is generated once but is free from code specific to the technical persistence component. In this class, application-specific functionality like, e.g., `checkCreditRating()` can be implemented manually. Code dependent from technical APIs and business application code are separated. Class `Customer` contains attributes (due to inheritance) and functionality which is closer to the idea of object-orientation. However, the use of implementation inheritance may be seen as problematic (see also the *fragile base class problem* [Szy02]).

6.5 Alternative 3: Separation Via Delegation

In the third alternative, A software (`Customer`) and R software (`CustomerGen`) are separated via delegation. From a design point of view, this is the cleanest solution.

However, in practical handling, this solution is most inconvenient. Since it adds a lot of workload to the application programmer, this solution is rarely used in practice.

6.6 Alternative 4: Separation Via Annotation

Modern programming languages like C# or Java 5 offer *annotations* as a built-in programming construct. Annotations are very similar to UML stereotypes. Hence, they can be depicted in diagrams, too. The fourth alternative mixes declarative and procedural programming. Annotations mark fields and methods that need generation of technical code. The application programmer can add application-specific functionality. However, generated code and manually programmed code is mixed. The next version of EJB, Version 3.0 [Sun1], is a good example for this approach. See, e.g., the column notation in [Sun2], Section 5.1.5.

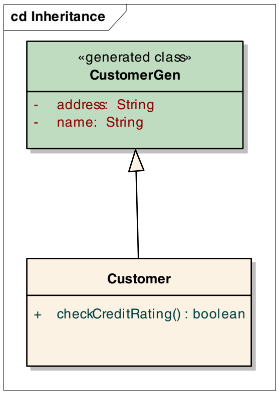


Fig. 7. Separation via inheritance

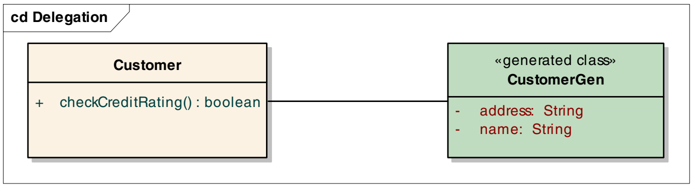


Fig. 8. Separation via delegation

6.7 Discussion

Which alternative to chose is a design decision which is to be taken by the chief designer according to project circumstances. In practice, Alternative 1 is usually chosen in projects with little application logic and Alternative 2 in more complex situations. With the advent of annotation features in modern programming languages like C# and Java 5, Alternative 4 will become most attractive since the application programmer has to understand only one level of model. The other three alternatives (1, 2, and 3) require that the programmer study the semantics of the PIM model, potentially the PSM model, code, and the relationships between the three models. The generated code usually builds a framework (indicated by the use of inheritance or delegation in our example). This induces another level of complexity with many known dangers [Ga97]. This is only acceptable for simple domains, like the data domains introduced above.

7 Conclusion

In conclusion, we postulate the following key statements:

1. MDD is an important technology since it helps to reduce the cost of software development and maintenance. Furthermore, it may increase the quality of the resulting systems. The cost reduction is due to avoiding tedious programming work. The increase in quality is due to unifying the coding styles of a potentially large number of programmers.
2. MDD is a proven technology with experience for over fifteen years.
3. The MDA initiative of the OMG is an important step. Standardization eases the integration of MDD tools and, thus, reduces the costs of setting up MDD environments. The standardization effort documents and advances the maturity of MDD.
4. MDD encompasses the generation as well as the interpretation of models.
5. It is naive to try to generate complete complex systems out of a single model.
6. One can and should identify domains for which MDD is appropriate. For those, domain-specific modelling languages should be used. In this paper, we have identified the MDD hot spot domains for business information systems. These domains are simple structures which are easy to understand.
7. Complex business application logic is best programmed manually in modern programming languages.
8. Business application code and technical code should be separated, particularly when using MDD. In this paper, we have presented principles and examples for such a separation of concerns.

MDD really works in practice, but it requires a lot of expertise and naive use may lead to disasters.

References

[Bro86] Frederick P. Brooks: No Silver Bullet - Essence and Accidents of Software Engineering: Information Processing 1986, ISBN No. 0444-7077-3, H. J. Kugler, Ed., Elsevia Science Publishers B.V. (North-Holland) IFIP 1986.

[CARE] CARE Technologies. Press Release Programmiermaschine: http://www.programmiermaschine.de/programmiermaschine.html

[Den93] Ernst Denert: “Dokumentenorientierte Software-Entwicklung”. Informatik Spektrum (1993) 16: S. 159 – 164.

[DS00] Denert, Ernst: Siedersleben, Johannes: Wie baut man Informationssysteme? Überlegungen zur Standardarchitektur. Informatik Spektrum 4/2000, pp 247-257.

[Ern04] Ernst, Andreas: Quasi-stellares Objekt; Objektbasierte Datenbankzugriffsschicht Quasar Persistence, Javamagazin 3/04, pp 85-88.

[Ga97]	Erich Gamma: 100 OO Frameworks, Pitfalls and Lessons Learned, 1997.
[GHJ+95]	Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Reading, Massachusetts, USA 1995.
[GS04]	Greenfield, J., Short, K.: Software Factories – Assembling Applications with Patterns, Models, Frameworks and Tools, Wiley 2004.
[HHS04]	Haft, Martin, Humm, Bernhard, Siedersleben, Johannes: Quasar Reference Interfaces for Business Information Systems. Technical Report sd&m Research. December, 2004.
[Hibernate]	Relational Persistence For Idiomatic Java: www.Hibernate.org
[Hum04]	Humm, Bernhard: Technische Open Source Komponenten implementieren die Referenzarchitektur Quasar. In: Helmut Eirund, Heinrich Jasper, Olaf Zukunft: ISOS 2004 - Informationsysteme mit Open Source, Proceedings GI-Workshop pp. 77-87. Gesellschaft für Informatik 2004.
[Kel94]	Wolfgang Keller: "Dokumentenorientierte Spezifikation objektorientierter Benutzeroberflächen". ONLINE ´94 - Congress VI – Innovative Softwaretechnologien: Neue Wege mit objektorientierten Methoden und Client/Server Architekturen.
[MOF]	Object Management Group: Meta-Object Facility V1.4. www.omg.org/mof/
[MM03]	Joaquin Miller, Jishnu Mukerji (Eds.), Object Management Group: "MDA Guide" Version 1.0.1. http://www.omg.org/docs/omg/03-06-01.pdf
[OMG]	Object Management Group. http://www.omg.org/
[OpenQuasar]	sd&m AG: Quasar components. http://www.openquasar.de
[Sch93]	Gero Scholz: „Maßgeschneiderte Software-Generatoren". Proceedings ONLINE 1993 Congress VI - C636.
[Sied04]	Siedersleben, Johannes: Moderne Software-Architektur – umsichtig planen, robust bauen mit Quasar. dpunkt Verlag. 2004.
[Sied03]	Siedersleben, Johannes (Hrsg.): Quasar: Die sd&m Standardarchitektur. Teile 1 und 2, 2. Auflage. sd&m Research, 2003.
[Sun1]	Sun Microsystems: EJB 3.0, Simplified API, Early Draft 2, http:// java.sun.com/ejb
[Sun2]	Sun Microsystems: EJB 3.0, Persistence API, Early Draft 2, http:// java.sun.com/ ejb
[Szy02]	Szyperski, C.: Component Software. Addison Wesley, 2002.
[Teu04]	Sören Teurich-Wagner: "MDA – Weg oder Irrweg?" In: B Rumpe, W Hesse (Hrsg) Tagungsband zur Modellierung 2004 (GI, 2004).
[TopLink]	Oracle Corp.: object relational mapper. http://www.oracle.com/technology/products/ias/TopLink
[UML]	Object Management Group: Unified Modeling Language V2.0. http://www.uml.org
[XMI]	Object Management Group: XML Metadata Interchange http://www.omg.org/technology/documents/formal/xmi.htm

Semantic Anchoring with Model Transformations*

Kai Chen, Janos Sztipanovits, Sherif Abdelwalhed, and Ethan Jackson

Institute for Software Integrated Systems, Vanderbilt University,
P.O. Box 1829 Sta. B., Nashville, TN 37235, USA
{kai.chen, janos.sztipanovits, sherif.abdelwalhed, ethan.jackson}
@vanderbilt.edu

Abstract. Model-Integrated Computing (MIC) is an approach to Model-Driven Architecture (MDA), which has been developed primarily for embedded systems. MIC places strong emphasis on the use of domain-specific modeling languages (DSML-s) and model transformations. A metamodeling process facilitated by the Generic Modeling Environment (GME) tool suite enables the rapid and inexpensive development of DSML-s. However, the specification of semantics for DSML-s is still a hard problem. In order to simplify the DSML semantics, this paper discusses semantic anchoring, which is based on the transformational specification of semantics. Using a mathematical model, Abstract State Machine (ASM), as a common semantic framework, we have developed formal operational semantics for a set of basic models of computations, called semantic units. Semantic anchoring of DSML-s means the specification of model transformations between DSML-s (or aspects of complex DSML-s) and selected semantic units. The paper describes the semantic anchoring process using the meta-programmable MIC tool suite.

1 Introduction

The Model-Driven Architecture (MDA) advocates a model-based approach for software development. Model-Integrated Computing (MIC) [27,24] is a domain-specific approach to MDA, which has been developed primarily for embedded systems. The MIC approach eases the complicated task of embedded system design by equipping developers with domain-specific modeling languages [25] tailored to the particular constraints and assumptions of their various application domains. A well-made DSML captures the concepts, relationships, integrity constraints, and semantics of the application domain and allows users to program imperatively and declaratively through model construction.

While a metamodeling process enables the rapid and inexpensive development of DSML syntax, the semantics specification for DSML-s remains a challenge problem. Transformational specification of semantics [9], gives us a chance

* This research was supported by the NSF Grant CCR-0225610 "Foundations of Hybrid and Embedded Software System".

to simplify the DSML semantics design. This paper exploits the transformational semantics specification approach for creating a semantic anchoring infrastructure [22]. This infrastructure incorporates a set of metaprogrammable MIC tools, including: the Generic Model Environment (GME) [4] for metamodeling, the Graph Rewriting and Transformation (GReAT) [2] tool for model transformation, the Abstract State Machines (ASM) [18,12], as a common semantic framework to define the semantic domain of DSML-s, and AsmL [1] – a high-level executable specification language based on the concepts of ASM for semantics specification.

The organization of this paper proceeds as follows: Section 2 describes the background for DSML specifications. Semantic anchoring is summarized in Section 3. In Section 4, we use a simple DSML that captures the finite state machine domain from Ptolemy II [5] as a case study to demonstrate the key steps in the semantic anchoring process. Conclusions and future work appear in Section 5.

2 Background: DSML Specification

A DSML can be formally defined as a 5-tuple $L = \langle A, C, S, M_S, M_C \rangle$ consisting of abstract syntax (A), concrete syntax (C), syntactic mapping (M_C), semantic domain (S) and semantic mapping (M_S) [28]. The syntax of a DSML consists of three parts: an abstract syntax, a concrete syntax, and a syntactic mapping. The abstract syntax A defines the language concepts, their relationships, and well-formedness rules available in the language. The concrete syntax C defines the specific notations used to express models, which may be graphical, textual, or mixed. The syntactic mapping, $M_C : C \rightarrow A$, assigns syntactic constructs to elements in the abstract syntax.

DSML syntax provides the modeling constructs that conceptually form an interface to the semantic domain. The semantics of a DSML provides the meaning behind each well-formed domain model composed from the syntactic modeling constructs of the language. For example, in MIC applications, the semantics of a domain model often prescribes the behavior that simulates an embedded system.

DSML semantics are defined in two parts: a semantic domain S and a semantic mapping $M_S : A \rightarrow S$ [21]. The semantic domain S is usually defined in some formal, mathematical framework, in terms of which the meaning of the models is explained. The semantic mapping relates syntactic concepts to those of the semantic domain. In DSML applications, semantics may be either structural or behavioral. The *structural semantics* describes the meaning of the models in terms of the structure of model instances: all of the possible sets of components and their relationships, which are consistent with the well-formedness rules, are defined by the abstract syntax. Accordingly, the semantic domain for structural semantics is defined by a *set-valued semantics*. The *behavioral semantics* may describe the evolution of the state of the modeled artifact along some time model. Hence, the behavioral semantics is formally captured by a mathematical framework representing the appropriate form of dynamics. In this paper, we focus on the behavioral semantics of a DSML.

3 Semantic Anchoring

Although DSML-s use many different notations, modeling concepts and model structuring principles for accommodating needs of domains and user communities, semantic domains for expressing basic behavior categories are more limited. A broad category of component behaviors can be represented by basic behavioral abstractions, such as Finite State Machine, Timed Automaton, Continuous Dynamics and Hybrid Automaton. This observation led us to the following strategy in defining behavioral semantics for DSML-s:

1. Define a set of minimal modeling languages $\{L_i\}$ for the basic behavioral abstractions and develop the precise specifications for all components of $L_i = \langle C_i, A_i, S_i, M_{Si}, M_{Ci}\rangle$. We use the term "semantic unit" to describe these basic modeling languages.
2. Define the behavioral semantics of an arbitrary $L = \langle C, A, S, M_S, M_C\rangle$ modeling language transformationally by specifying the $M_A : A \rightarrow A_i$ mapping. The $M_S : A \rightarrow S$ semantic mapping of L is defined by the $M_S = M_{Si} \circ M_A$ composition, which indicates that the semantics of L is anchored to the S_i semantic domain of the L_i modeling language.

The tool architecture supporting the semantic anchoring process above is shown in Figure 1. The GME tool suite [4] is used for defining the abstract syntax, A, for an $L = \langle C, A, S, M_S, M_C\rangle$ DSML using UML Class Diagrams [7] and OCL as metalanguage [8]. The $L_i = \langle C_i, A_i, S_i, M_{Si}, M_{Ci}\rangle$ semantic unit is defined as an AsmL specification [1] in terms of (a) an AsmL Abstract Data Model (which corresponds to the A_i, abstract syntax specification of the modeling language defining the semantic unit in the AsmL framework), (b) the S_i, semantic domain (which is implicitly defined by the ASM mathematical framework), and (c) the M_{Si}, semantic mapping (which is defined as a model interpreter written in AsmL).

The $M_A : A \rightarrow A_i$ semantic anchoring of L to L_i is defined as a model transformation using the GReAT tool suite [2]. The abstract syntax A and A_i

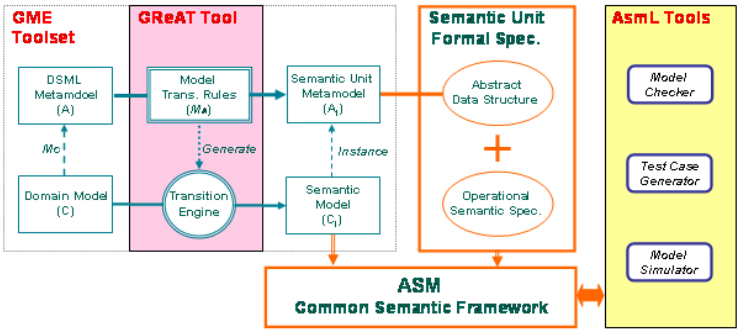


Fig. 1. Tool Architecture for DSML Design throughs Semantic Anchoring

are expressed as metamodels. Connection between the GME-based metamodeling environment and the AsmL environment is provided by a simple syntax conversion. Since the GReAT tool suite generates a model translation engine from the meta-level specification of the model transformation [23], any legal domain model defined in the DSML can be directly translated into a corresponding AsmL data model and can be simulated by using the AsmL native simulator. In the following, we give explanation of our methodology and the involved tools.

3.1 Formal Framework for Specifying Semantic Units

Semantic anchoring requires the specification of semantic units in a formal framework using a formal language, which is not only precise but also manipulable. The formal framework must be general enough to represent all three components of the $M_S : A \rightarrow S$ specification; the abstract syntax, A, with set-valued semantics, the S semantic domain to represent the dynamic behavior and the mapping between them. We select Abstract State Machine (ASM) as a formal framework for the specification of semantic units.

Abstract State Machine (ASM), formerly called Evolving Algebras [18], is a general, flexible and executable modeling structure with well-defined semantics. General forms of behavioral semantics can be encoded as (and simulated by) an abstract state machine [12]. ASM is able to cover a wide variety of domains: sequential, parallel, and distributed systems, abstract-time and real-time systems, and finite- and infinite-state domains. ASM has been successfully used to specify the semantics of numerous languages, such as C [19], Java [14], SDL [17] and VHDL [13]. In particular, the International Telecommunication Union adopted an ASM-based formal semantics definition of SDL as part of SDL language definition [6].

The Abstract State Machine Language, AsmL [1], developed by Microsoft Research, makes writing ASM specifications easy within the .NET environment. AsmL specifications look like pseudo-code operating on abstract data structures. As such, they are easy for programmers to read and understand. A set of tools is also provided to support the compilation, simulation, test case generation and model checking for AsmL specifications. The fact that there exists plentiful supporting tools for AsmL specifications was a important reason for us to select AsmL over other formal specification languages, such as Z [16], tagged signal model [26] and Reactive Modules [11]. A detailed introduction to ASM and AsmL is beyond the scope of this paper, but readers can refer to other papers [1,12,18].

3.2 Formal Framework for Model Transformation

We use model transformation techniques as a formal approach to specify the $M_A : A \rightarrow A_i$ mapping between the abstract syntax of a DSML and the abstract syntax of the semantic unit. Based on our discussion above, the abstract syntax A of the DSML is defined as a metamodel using UML class diagrams and OCL, and the A_i abstract syntax of the semantic unit is an Abstract Data Model expressed using the AsmL data structure. However, the specification of the M_A transformation requires that the domain and codomain of the transformation is expressed in the same language. In our tool architecture, this common language is the abstract syntax metamodeling language (UML class diagrams and OCL), since the GReAT tool suite is based on this formalism.

This choice requires building a UML/OCL-based metamodeling interface for the Abstract Data Model used in the AsmL specification of the semantic unit. One possible solution is to define a UML/OCL metamodel that captures the abstract syntax of the generic AsmL data structures. The other solution is to construct a metamodel that captures only the exact syntax of the AsmL Abstract Data Model of a particular semantic unit. Each solution has its own advantages and disadvantages. In the first solution, different semantic units can share the same metamodel and the same AsmL generator can be used to generate the data model in the native AsmL syntax. The disadvantage is that the model transformation rules and the AsmL specification generator are more complicated. Figure 2 shows a simplified version of the metamodel of generic AsmL data structures as it appears in the GME metamodeling environment. In the second solution, a new metamodel needs to be constructed for different semantic units, but the transformation rules are simpler and more understandable. Since the metamodel construction is easier compared with the specification of model transformation rules, we selected the second solution in our current work. We will present a metamodel example using this approach in section 4.4.

The $M_A : A \rightarrow A_i$ semantic anchoring is specified by using the Unified Model Transformation (UMT) language of the GReAT tool suite [23]. UMT itself is a

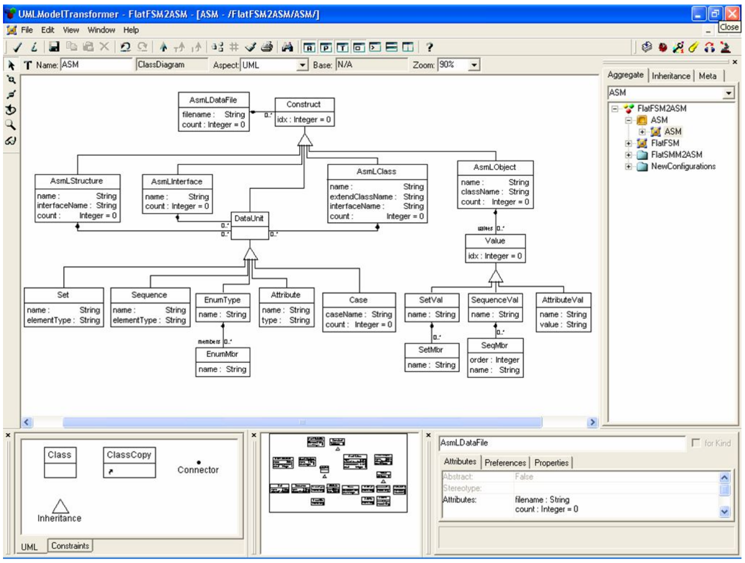


Fig. 2. Metamodel for a Set of AsmL Data Structures

DSML and the transformation M_A can be specified graphically using the GME tool. The GReAT tool uses GME and allows users to specify model-to-model transformation algorithms as graph transformation rules between metamodels. The transformation rules between the source and the target metamodels form the semantic anchoring specifications of a DSML. The GReAT engine can execute these transformation rules and transform any allowed domain model to an AsmL model stored in an XML format. Then the AsmL specification generator parses the XML file, performs specification rewriting and generates data model in the native AsmL syntax. Note that UMT provides designers with certain modeling constructs (e.g. "any match") to specify non-deterministic graph transformation algorithms. However, we can always achieve a unique semantic anchoring result by using only the UMT modeling constructs that do not cause the non-determinism.

4 Semantic Anchoring Case Study: FSM Domain in Ptolemy II

We have applied the semantic anchoring method and tool suite to design several DSML-s, including one patterned after the finite state machine (FSM) domain in Ptolemy II [5], the MATLAB Stateflow [20], and the IF timed automata based modeling language [15]. The detailed implementation can be downloaded from [3]. We use the FSM domain from Ptolemy II as a case study to illustrate the process described above.

4.1 The FSM Domain in Ptolemy

The Ptolemy FSM domain was proposed by Edward Lee with the name *charts [10] in 1999. It allows the composition of hierarchical FSMs with a variety of concurrency models. For simplicity, we define a DSML called the FSM Modeling Language (FML) which only supports Ptolemy-style hierarchical FSMs. For a detailed description of *charts and the hierarchical FSMs in Ptolemy II, readers may refer to [5,10].

4.2 The Abstract Syntax Definition for FML

Figure 3 shows a UML class diagram for the FML metamodel as represented in GME. The classes in the UML class diagram define the domain modeling concepts. For example, the *State* class denotes the FSM domain concept of state. Instances of the *State* class can be created in a domain model to represent the states of a specific FSM. Note that the *State* class is hierarchical: each *State* object can contain another state machine as a child in the hierarchy.

A set of OCL constraints is added to the UML class diagram to specify well-formedness rules. For example, the constraint,

```
self.parts(State)→size>0 implies
self.parts(State)→select(s:State|s.initial)→size=1,
```

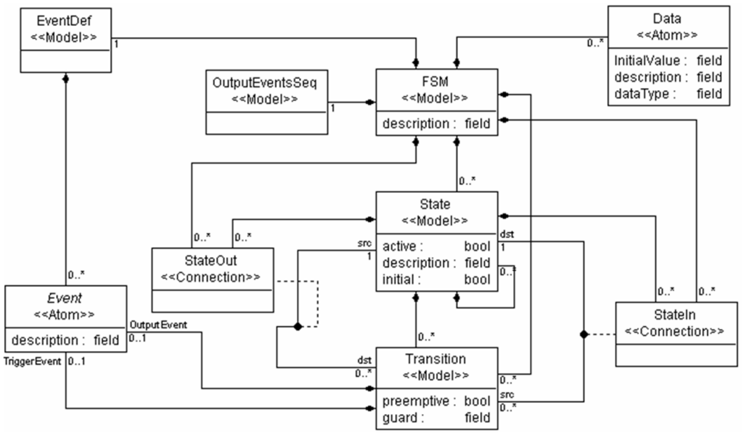


Fig. 3. A UML Class Diagram for the FML Metamodel

is attached to the *FSM* class. It specifies that if a *FSM* object has child states, exactly one child state must be the initial state. This is a constraint in Ptolemy II FSM domain.

Visualizations for instances of classes also need to be specified in the metamodel, so that an icon in a domain model will denote an instance of the corresponding class in the metamodel. In GME, this is usually done by setting a metamodel class's "Icon" attribute to the name of the desired bitmap.

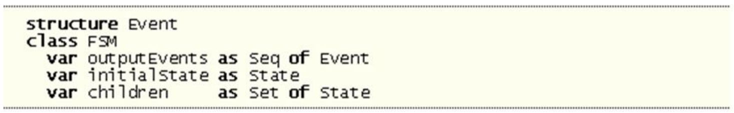
4.3 Semantic Unit Specifications for FML

An appropriate semantic unit for FML should be generic enough to express the behavior of all syntactically correct FSMs. Since our purpose in this paper is restricted to demonstrate the key steps in semantic anchoring, we do not investigate the problem of identifying a generic semantic unit for hierarchical state machines. We simply define a semantic unit, which is rich enough for FML.

The semantic unit specification includes two parts: an Abstract Data Model and a Model Interpreter defined as operational rules on the data structures. Whenever we have a domain model in AsmL (which is a specific instance of the Abstract Data Model), this domain model and the operational rules compose an abstract state machine, which gives the model semantics. The AsmL tools can simulate its behavior, perform the test case generation or perform model checking. Since the size of the full semantic unit specification is substantial, we can only show a part of the specifications together with some short explanations. Interested readers can download the full specifications from [3].

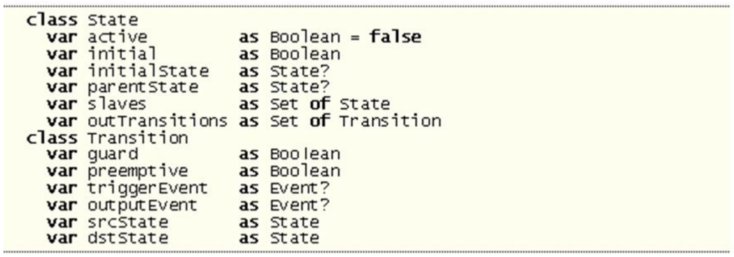
Constructing Abstract Data Model for FML. In this step, we specify an Abstract Data Model using AsmL data structures, which will correspond to the semantically meaningful modeling constructs in FML. The Abstract Data Model does not need to capture every details of the FML modeling constructs, since some of them are only semantically-redundant. The semantic anchoring (i.e. the mapping between the FML metamodel and the Abstract Data Model) will map the FML abstract syntax onto the AsmL data structures that we specify below.

Event is defined as an AsmL abstract data type *structure*. It may consist of one or more fields via the AsmL *case* construct. The keyword *structure* in AsmL declares a new type of compound value. In AsmL, classes contain instance variables and are the only way to share memory. Structures contain fields and do not share memory. Note that each AsmL language construct has its mathematical meaning in ASM. Readers can refer to [29] for their relationships. These fields are model-dependent specializations of the semantic unit, which give meaning to different types of events. The AsmL class *FSM* captures the top-level of the hierarchical state machine. The field *outputEvents* is an AsmL *sequence* recording the chronologically-ordered model events generated by the FSM. The field *initialState* records the start state of a machine. The *children* field is an AsmL set that records all state objects which are the top-level children of the state machine.



```
structure Event
class FSM
  var outputEvents as Seq of Event
  var initialState as State
  var children     as Set of State
```

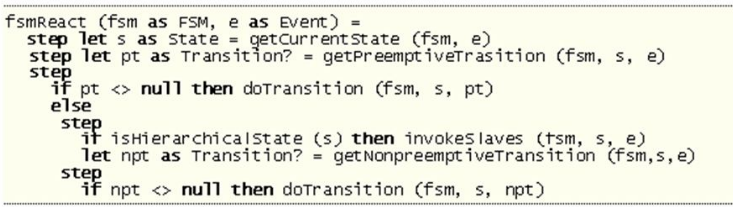
State and Transition are defined as first-class types. Note that the variable field *initalState* of the *State* class records the start state of any child machine contained within a given *State* object. The *initalState* will be undefined whenever a state has no child states. This possibility forces us to add the ? modifier to express that the value of the field may be either a *State* instance or the AsmL *undef* value. For a similar reason, we add the ? modifier after several other types of variable fields.



```
class State
  var active         as Boolean = false
  var initial        as Boolean
  var initialState   as State?
  var parentState    as State?
  var slaves         as Set of State
  var outTransitions as Set of Transition
class Transition
  var guard          as Boolean
  var preemptive     as Boolean
  var triggerEvent   as Event?
  var outputEvent    as Event?
  var srcState       as State
  var dstState       as State
```

Behavioral Semantics for FML. We are now ready to specify the behavioral semantics for FML as operational rules, which can interpret the AsmL data structures defined above. Due to the space limitation, we show only two operational rules here.

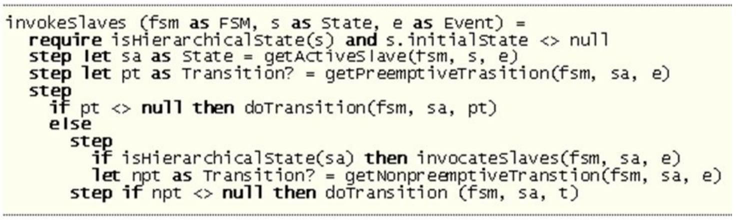
Top-Level FSM Operations. A *FSM* instance waits for input events. Whenever an allowed input event arrives, the *FSM* instance reacts in a well-defined manner by updating its data fields and activating enabled transitions. To avoid non-determinism, the Ptolemy II FSM domain defined its own priority policy for transitions, which supports both the hierarchical priority concept and preemptive interrupt. The operational rule *fsmReact* specifies this reaction step-by-step. Note that the AsmL keyword *step* introduces the next atomic step of the abstract state machine in sequence. The operations specified within a given step all occur simultaneously.



```
fsmReact (fsm as FSM, e as Event) =
  step let s as State = getCurrentState (fsm, e)
  step let pt as Transition? = getPreemptiveTrasition (fsm, s, e)
  step
    if pt <> null then doTransition (fsm, s, pt)
    else
     step
       if isHierarchicalState (s) then invokeSlaves (fsm, s, e)
       let npt as Transition? = getNonpreemptiveTransition (fsm,s,e)
     step
       if npt <> null then doTransition (fsm, s, npt)
```

First, the rule determines the current state, which might be an initial state. Next, it checks for enabled preemptive transitions from the current state. If one exists, then the machine will take this transition and end the reaction. Otherwise, the rule will first determine if the current state has any child states. If it does, then the rule will invoke the child states of the current state. Next, it checks for enabled non-preemptive transitions from the current state. If one exists, the rule will take this transition and end this reaction. Otherwise, it will do nothing and end this reaction.

Invoke Slaves. The operational rule *invokeSlaves* describes the operations required to invoke the child machine in a hierarchical state. The AsmL construct *require* is used here to assert that this state should be a hierarchical state, and it should have a start state in its child machine. The rule first determines the active state in the child machine. The rest of this operational rule is the same as the *fsmReact* rule. The similarity between the reactions of the top-level state machine and any child machine facilitates the Ptolemy II style composition of different models of computations.



```
invokeSlaves (fsm as FSM, s as State, e as Event) =
  require isHierarchicalState(s) and s.initialState <> null
  step let sa as State = getActiveSlave(fsm, s, e)
  step let pt as Transition? = getPreemptiveTrasition(fsm, sa, e)
  step
    if pt <> null then doTransition(fsm, sa, pt)
    else
      step
        if isHierarchicalState(sa) then invocateSlaves(fsm, sa, e)
        let npt as Transition? = getNonpreemptiveTranstion(fsm, sa, e)
      step if npt <> null then doTransition (fsm, sa, t)
```

4.4 Semantic Anchoring Specifications for FML to the Semantic Unit

Having the abstract syntax of FML and an appropriate semantic unit specified, we are now ready to describe the semantic anchoring specifications for FML. We use UMT, a language supported by the GReAT tool, to specify the model transformation rules between the metamodel of FML (Figure 3) and the metamodel for the semantic unit shown in Figure 4.

The semantic anchoring specifications in UMT consist of a sequence of model transformation rules. Each rule is finally expressed using pattern graphs. A pattern graph is defined using associated instances of the modeling constructs defined in the source and destination metamodels. Objects in a pattern graph can play three different roles as follows:

1. *bind*: Match object(s) in the graph.
2. *delete*: Match objects(s) in the graph, then, remove the matched object(s) from the graph.
3. *new*: Create new objects(s) provided all of the objects marked *Bind* or *Delete* in the pattern graph match successfully.

The execution of a model transformation rule involves matching each of its constituent pattern objects having the roles *bind* or *delete* with objects in the input and output domain model. If the pattern matching is successful, each combination of matching objects from the domain models that correspond to the pattern objects marked *delete* are deleted and each new domain objects that correspond to the pattern objects marked *new* are created.

We give an overview of the model transformation algorithm with a short explanation for selected role-blocks below. The transformation rule-set consists of the following steps:

1. Start by locating the top-level state machine in the input FML model; create an AsmL *FSM* object and set its attribute values appropriately.

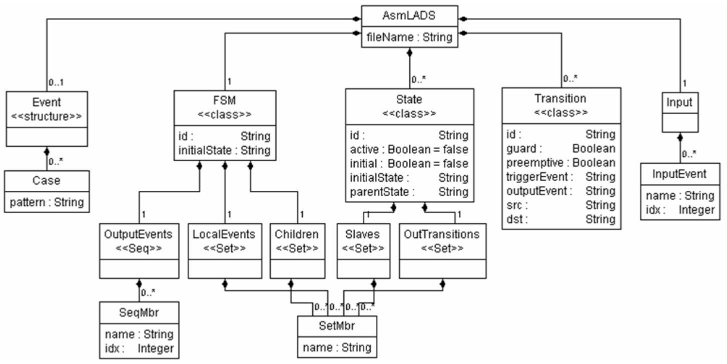


Fig. 4. Metamodel Capturing AsmL Abstract Data Structures for FML

2. *Handle Events*: Match the event definitions in the input model and create the corresponding variants through the *Case* construct in *Event*.
3. *Handle States*: Navigate through the FML *FSM* object; map its child *State* objects into instances of AsmL *State* class, and set their attribute values appropriately. Since the *State* in FML has a hierarchical structure, the transformation algorithm needs to include a loop to navigate the hierarchy of *State* objects.
4. *Handle Transition*: Navigate the hierarchy of the input model; create an AsmL *Transition* object for each matched FML *Transition* object and set its attribute values appropriately.

Figure 5 shows the top-level transformation rule that consists of a sequence of sub-rules. These sub-rules are linked together through ports within the rule boxes. The connections represent the sequential flow of domain objects to and from rules. The ports *FSMIn*, and *AsmLIn* are input ports, while ports *FSMOut* and *AsmLOut* are output ports. In the top-level rule, *FSMIn* and *AsmLIn* are bound to the top-level state machine in the FSM model that is to be transformed, and the root object (a singleton instance of *AsmLADS*) in the semantic data model that is to be generated, respectively. The four key steps in the transformation algorithm, as described above, are corresponding to the four sub-rules contained in the top level rule.

The figure also shows a hierarchy, i.e., a sub-rule may be further decomposed into a sequence of sub-rules. The *CreateStateObjects* rule outlines a graphical algorithm which navigates the hierarchical structure of a state machine. It starts from the root state, does the bread-first navigation to visit all child state objects and creates corresponding AsmL *State* objects.

Figure 6 shows the *SetAttributes* rule. This rule sets the attribute values for the newly created AsmL *State* object. First, the sub-rule *SetInitialState* checks whether the current FML *State* object is a hierarchical state and has a start state. If it has a start state, set the value of the attribute *initialState* to this start state. Otherwise set the value to *null*. Then, the sub-rule *SetSlaves* searches for all hierarchically-contained child states in the current state and adds them as members into the attribute *Slave* whose type is a set. Finally, the transitions out

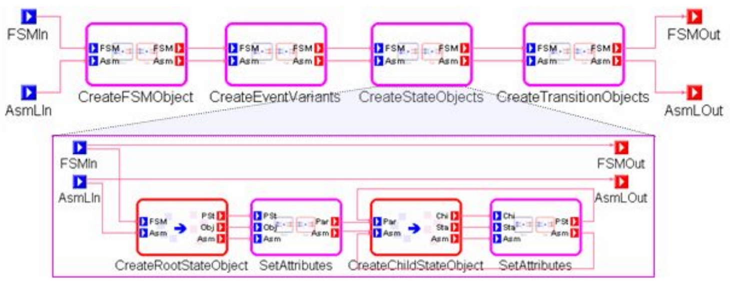


Fig. 5. Top-level model transformation rule

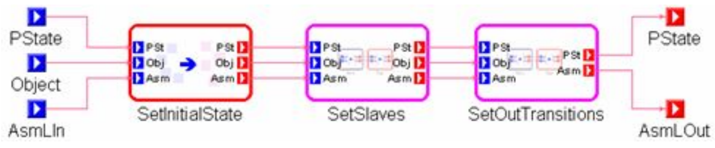
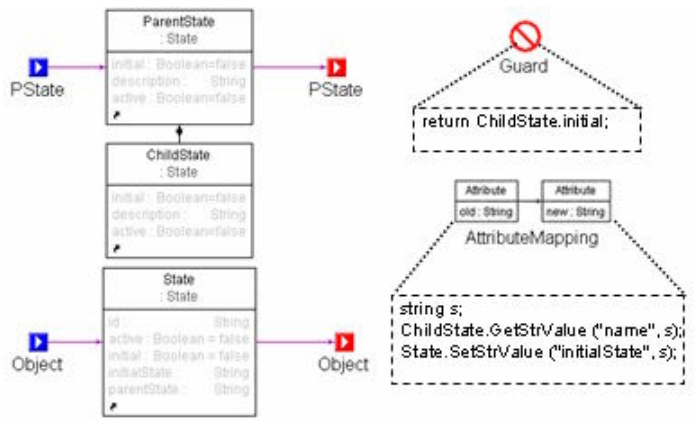


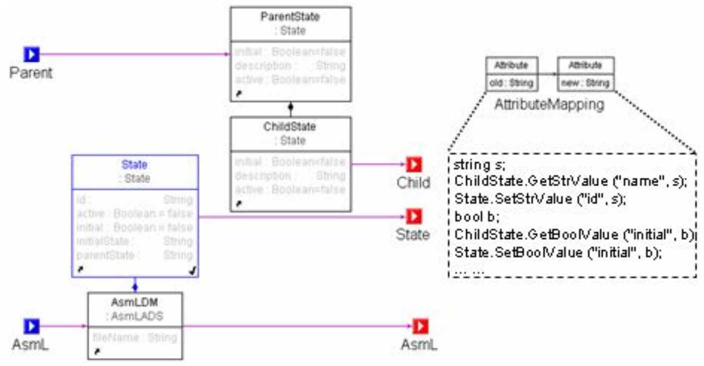
Fig. 6. Model Transformation Rule: *SetAttributes*



```
return ChildState.initial;
```

```
string s;
ChildState.GetStrValue ("name", s);
State.SetStrValue ("initialState", s);
```

Fig. 7. Model Transition Rule: *SetInitialState*



```
string s;
ChildState.GetStrValue ("name", s);
State.SetStrValue ("id", s);
bool b;
ChildState.GetBoolValue ("initial", b);
State.SetBoolValue ("initial", b);
... ...
```

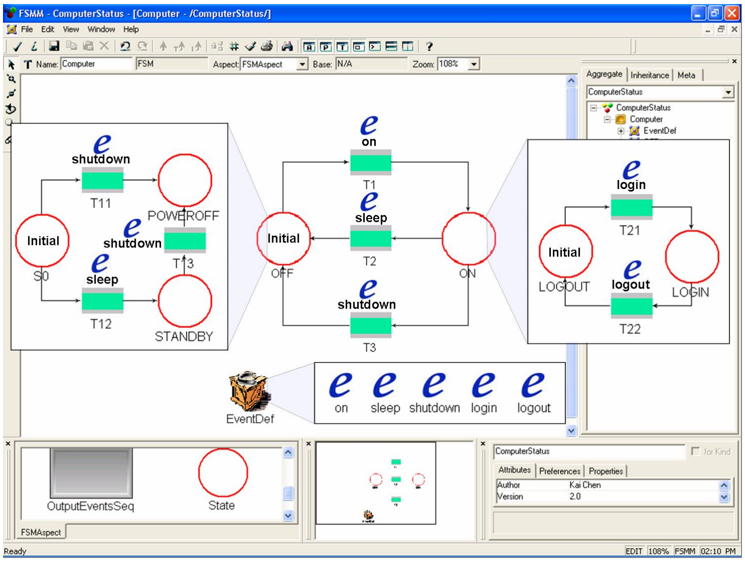
Fig. 8. Model Transition Rule: *CreateChildStateObject*

from the current state are added as members to the attribute *OutTransitions* by the sub-rule *SetOutTransitions*.

The final contents of model transformation rules are pattern graphs that are specified in UML class diagrams. Figure 7 shows a part of the *SetInitial-State* rule, which is a pattern graph. This rule features a GReAT *Guard* code block and a GReAT *AttributeMapping* code block. This rule is executed only if the graph elements match and the *Guard* condition evaluates to true. The *AttributeMapping* block includes code for reading and writing object attributes.

The *CreateChildStateObejct* rule, shown in Figure 8, creates a new AsmL *State* object when a FML child State object is matched. It also enables the hierarchy navigation. Through a loop specified in the *CreateStateObjects* rule (Figure 5), the child *State* object output by the *Child* port will come back as an input object to the *Parent* port.

In the semantic anchoring process, the GReAT engine takes a legal FML domain model, executes the model transformation rules and generates an AsmL


FSMM - ComputerStatus - [Computer - /ComputerStatus/]

Name: Computer FSM Aspect: FSMAspect Base: N/A Zoom: 108%

ComputerStatus

Attributes: Author Kai Chen; Version 2.0

Ready EDIT 108% FSMM 02:10 PM

Fig. 9. A Hierarchical FSM model: *ComputerStatus*

```
structure Event
   case on
   case sleep
   case shutdown
   case login
   case logout
ComputerStatus = new FSM([], OFF, {ON,OFF})
OFF       = new State(true, S0, null, {S0, POWEROFF, STANDBY}, {T1})
ON        = new State(false, LOGOUT, null, {LOGOUT, LOGIN}, {T3,T2})
S0        = new State(true, null, OFF, {}, {T12,T11})
POWEROFF  = new State(false, null, OFF, {}, {})
STANDBY   = new State(false, null, OFF, {}, {T13})
LOGOUT    = new State(true, null, ON, {}, {T21})
LOGIN     = new State(false, null, ON, {}, {T22})
T1  = new Transition(true, false, Event.on, null, OFF, ON)
T2  = new Transition(true, false, Event.sleep, null, ON, OFF)
T3  = new Transition(true, false, Event.shutdown, null, ON, OFF)
T11= new Transition(true, false, Event.shutdown, null, S0, POWEROFF)
T12= new Transition(true, false, Event.sleep, null, S0, STANDBY)
T13= new Transition(true, false, Event.shutdown, null,STANDBY,POWEROFF)
T21= new Transition(true, false, Event.login, null, LOGOUT, LOGIN)
T22= new Transition(true, false, Event.logout, null, LOGIN, LOGOUT)
```

Fig. 10. Part of the AsmL Data Model Generated from the *ComputerStatus* Model

data model. As an example, we design a simple hierarchical *FSM* model in the GME modeling environment (Figure 9), which simulates the status of a computer. An XML file storing the AsmL data model is generated through the semantic anchoring process. We developed an AsmL specification generator, which can parse this XML file and generate the data model in native AsmL syntax as shown in Figure 10. The newly created AsmL data model plus the previously-defined AsmL semantic domain specifications compose an abstract state machine that gives the semantics for the FSM model *ComputerStatus*. With these specifications, the AsmL tools can simulate the behavior, do the test case generation and model checks. For more information about the AsmL supported analysis, see [1].

5 Conclusion and Future Work

This paper proposes a rapid and formal DSML design methodology, which integrates the semantic anchoring method and the metamodeling process. As the example showed, combining operational specification of semantic units with the transformational specification of DSML-s has the potential for improving significantly the precision of DSML specifications. We expect that substantial further effort is required to identify the appropriate set of semantic units and the best formal framework, which is general enough to cover a broad range of models of computations and can integrate both operational and denotational semantic specifications. We are now working on specifying a semantic unit that can capture the common semantics for varied real-time system modeling languages. An interesting area for further research is use cases for semantic units. This may include the automatic generation of model translators that confirm the operational semantic captured in the semantic unit and offer semantically well founded tool integration and tool verification technology.

References

1. The Abstract State Machine Language. `www.research.microsoft.com/fse/asml`.
2. Graph Rewriting and Transformation. `www.isis.vanderbilt.edu/Projects/mobies`.
3. Link for semantic anchoring tool suite. `www.isis.vanderbilt.edu/SAT`.
4. The Generic Modeling Environment: GME. `www.isis.vanderbilt.edu/Projects/gme`.
5. The Ptolemy II. `www.ptolemy.eecs.berkeley.edu/ptolemyII`.
6. ITU-T recommendation Z.100 annex F: SDL formal semantics definition. International Telecommunication Union, Geneva, 2000.
7. OMG unified modeling language specification version 1.5. Object Management Group document, 2003. formal/03-03-01.
8. UML 2.0 OCL final adopted specification. Object Management Group document, 2003. ptc/03-10-14.
9. A. Maggiolo-Schettini and A. Peron. Semantics of full statecharts based on graph rewriting. In *LNCS*, pages 265–279. Springer-Verlag, 1994.

10. Alain GiraltB, Bilung Lee and E. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, 18(6), 1999.
11. R. Alur and T. A. Henzinger. Reactive modules. *Form. Methods Syst. Des.*, 15(1):7–48, 1999.
12. E. Boerger and R. Staerk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer, 2003.
13. E. Borger, U. Glasser, and W. Muller. *Formal Semantics for VHDL*, chapter Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines, pages 107–139. Kluwer Academic Publishers, 1995.
14. E. Borger and W. Schulte. A programmer friendly modular definition of the semantics of java. In *Formal Syntax and Semantics of Java, LNCS*, volume 1523, pages 353–404. Springer-Verlag, 1999.
15. M. Bozga, S. Graf, I. Ober, and J. Sifakis. Tools and applications II: The IF toolset. In *Proceedings of SFM'04, LNCS*, volume 3185. Springer-Verlag, 2004.
16. A. Diller. *Z: an Introduction to Formal Methods.* John Wiley & Sons Ltd., second edition, 1994.
17. U. Glasser and R. Karges. Abstract state machines semantics of SDL. *Journal of University Computer Science*, 3(12):1382–1414, 1997.
18. Y. Gurevich. *Specification and Validation Methods*, chapter Evolving Algebras 1993: Lipari Guide, pages 9–36. Oxford University Press.
19. Y. Gurevich and J. Huggins. The semantics of the C programming languages. In *Computer Science Logic'92*, pages 274–308. Springer-Verlag, 1993.
20. G. Hamon and J. Rushby. An operational semantics for stateflow. In *Fundamental Approaches to Software Engineering: 7th International Conference*, pages 229–243. Springer-Verlag, 2004.
21. D. Harel and B. Rumpe. Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer*, 37(10), 2004.
22. Kai Chen, J. Sztipanovits, S. Neema, M. Emerson and S. Abdelwahed. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *5th ACM International Conference on Embedded Software (EMSOFT'05)*, 2005.
23. G. Karsai, A. Agrawal, and F. Shi. On the use of graph transformations for the formal specification of model interpreters. *Journal of Universal Computer Science*, 9(11):1296–1321, 2003.
24. G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. In *Proceedings of the IEEE*, volume 91, pages 145–164, 2003.
25. A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *IEEE Computer*, 34(11):44–51, 2001.
26. E. Lee and A. Sangiovanni-Vincentelli. A denotational framework for comparing models of computation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 17(12), 1998.
27. J. Sztipanovits and G. Karsai. Model-integrated computing. *IEEE Computer*, 30(4):110–111, 1997.
28. T. Clark, A. Evans, S. Kent and P. Sammut. The MMF approach to engineering object-oriented design languages. In *Workshop on Language Descriptions, Tools and Applications*, 2001.
29. Yuri Gurevich, Benjamin Rossman and W. Schulte. Semantics essence of AsmL, March 2004. MSR-TR-2004-27.

On Some Properties of Parameterized Model Application

Alexis Muller[1], Olivier Caron[1], Bernard Carré[1], and Gilles Vanwormhoudt[1,2]

[1] Laboratoire d'Informatique Fondamentale de Lille,
UMR CNRS 8022 - INRIA Jacquard Project,
Université des Sciences et Technologies de Lille,
59655 Villeneuve d'Ascq cedex, France
[2] GET/ENIC Telecom Lille I
`{mullera, carono, carre, vanwormh}@lifl.fr`

Abstract. Designing Information Systems (IS) is a complex task that involves numerous aspects, being functional or not. A way to achieve this is to consider models as generic pieces of design in order to build a complete IS. Model composition provides a way to combine models and model parameterization allows the reuse of models in multiple contexts. In this paper, we focus on the use of parameterized models in model driven engineering processes. We outline the needs to compose parameterized models and apply them to a system according to alternative and coherent ordering rules. Such building processes raise open issues: Is the result influenced by the order of applications ? Can we compose independent parameterized models ? Is it possible to define composition chains and find equivalent ones that express the same resulting model ? These requirements are formalized through an apply operator. This operator guarantees properties which can help in the formulation of model driven system construction methodologies. Finally, we briefly describe a modelling tool that supports processes based on this operator.

1 Introduction

Illustrated by new approaches of software development [13,1,12], models are gaining more and more importance in the software development lifecycle. There is a growing need to use them as concrete artefacts [11] through operations like projections, translations or constructions. Projection techniques mainly aim to transpose a model from a technological space (UML for example) to another one (like EJB or CORBA). Translation techniques allow to express the same model in another language (UML to XMI for example). Finally, construction techniques ambition is to produce new models from existing ones.

Among these construction techniques, composition techniques, which permit the building a model from a set of smaller ones, are widely used. Indeed, in spite of new development approaches, systems and thus their models become more complex and bigger. It is thus necessary to deal with this complexity by providing the decomposition of such systems and their models.

Building new systems by composing prefabricated and validated models promises a quicker design of more reliable systems. It is possible to identify in any information system some concerns that should be applied to other systems [8,10,14]. Nevertheless, their models are made according to a particular system so that they are hard to reuse in order to build new systems. The design of reusable models calls for the usage of some sort of parameterized models or model templates [10,9,18,19]. As far as standardization is concerned, UML2 defines the notion of template [2] which allows representing such generic models as packages parameterized by model elements. In [5], we have formalized the UML2 template binding relationship and introduced OCL rules for checking the correct matching between the required model (specified in the template signature) and a model constructed using this template. This relationship is independent from any construction process.

In this paper, we focus on how to define model driven engineering construction processes with such parameterized models. This needs to express complex compositions of parameterized models which must be applied in a coherent way. Such building processes raise open issues: Is the result influenced by the application order? Can we compose independent parameterized models? Is it possible to define composition chains and find equivalent ones that express the same resulting model?

To support such processes, we introduce an operator (*apply*) to express the application of a template to an existing model. This operator allows to specify how to obtain a model from an existing one by the application and composition of generic ones. It is interesting to note that generic models are models so that template applications can be combined to design richer ones. The next section shows the needs for such an operator and specify its expected properties. Then the third section presents a formalization of its semantics and proves these properties. Section 4 discusses related works about composition and parameterization of models. Finally, we briefly describe a modelling tool that supports processes based on this operator and provides strategies to transform composition of parameterized models into platform specific models before concluding.

2 Applying Parameterized Models

It is possible to use UML 2 template packages to represent parameterized models. For example, let us consider a set of parameterized models designed for resource management systems (inspired from [8] and [21]) where each model provides a useful function such as searching, stock management and resource allocation.

Figure 1 shows a model offering resource management functionalities related to a stock (add, delete, and transfer operations). This model is specified by a template package owning a class diagram providing these functionalities. Elements required for its usage are exposed in the template signature. In our example, the required elements are the classes *Stock* and *Resource*, some of their corresponding properties *identifier* and *ref*, and the association *in*. This element set forms a unique parameter corresponding to the model structure required by the tem-

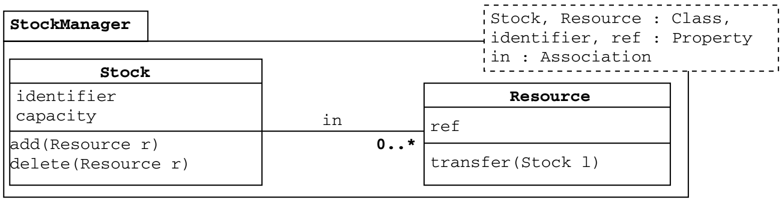


Fig. 1. Resource Management Template

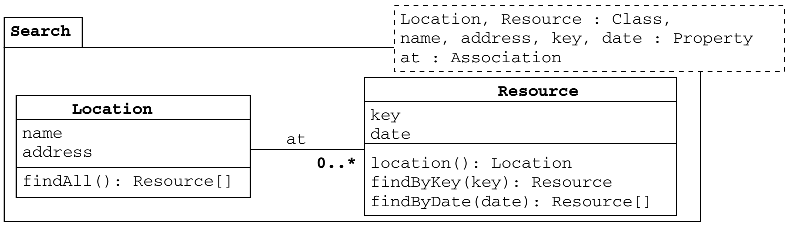


Fig. 2. Search Template

plate to be applied. The other elements correspond to the specific elements of the functionality defined by the template. Those specific elements will be added to which the template will be applied.

Figures 2, 3, and 4 respectively illustrate generic models for searching, resource allocation, and counting. These template examples show that elements of a required model can be either properties, operations, associations or classes.

To illustrate the use of these templates, let us take the example of a car hiring system. Figure 5 shows the primary model of this system. This base describes the structure of the different domain classes used by the system (here *Car*, *Agency*, and *Client*).

The desired system must be able to search a specific car or a specific client, and also to manage the different car allocations. To achieve this, we will use the generic models described before.

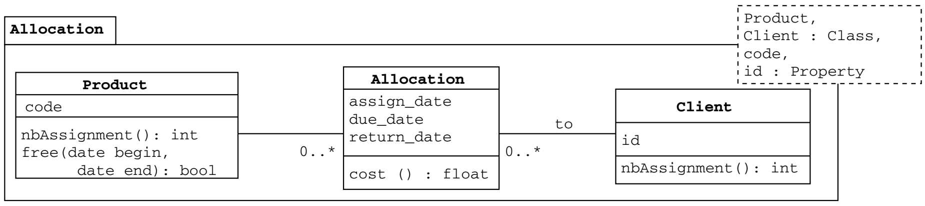


Fig. 3. Resource Allocation Template

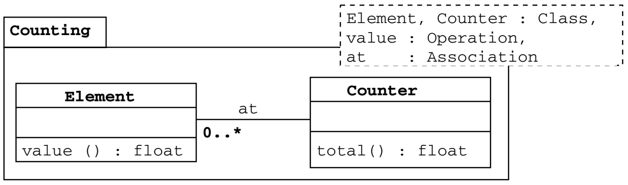


Fig. 4. Counting Template

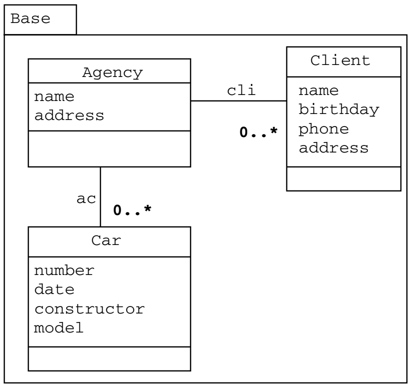


Fig. 5. The Base System

Thus, we need to specify an assembly of these models. For that, we introduce a parameterized model application operator called *apply*. Figure 6 shows how the *Stock Manager* template is applied to the base system. The *apply* operator is specified via an UML stereotyped dependency <<*apply*>> between the template source model and the system target model which establishes correspondence relationships between their respective model elements. This dependency includes the substitution of formal parameters (source model elements) by effective parameters (target model elements). The effective parameters must form a model that matches the required model of the template.

A formulation of a resulting system where corresponding elements are linked by <<trace>> dependencies[1] is shown in Figure 7. Our formulation does not impose any mapping for these <<trace>> dependencies. It is possible, for example, to merge linked elements or to use split representation mechanisms. We already have studied some of these targeting strategies [16,6].

These sketches show how to obtain an extended model from an existing one by the application of some parameterized models. To gain more reuse, composition of parameterized models should be supported too, in order to design complex generic models from simpler ones. This facility is illustrated in Figure 8(a) where

[1] The <<trace>> dependency is a standard UML2 relationship. It is used to link elements that represent the same concept.

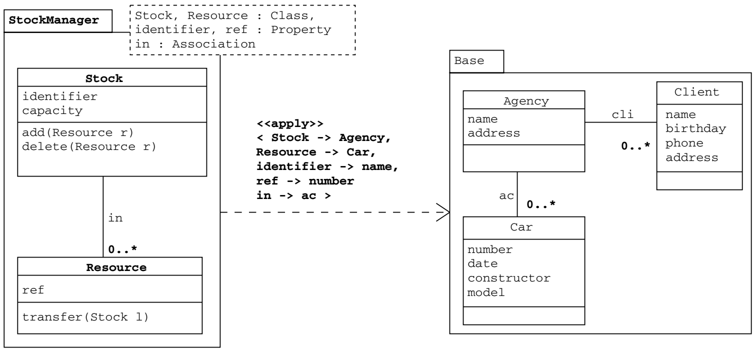


Fig. 6. Applying Stock Manager to Car Hiring System

the *Search* template applied to the *StockManager* template allows to build a new generic model (see Figure 8(b)). The set of parameters of this new model is determined by the union of the target model parameters (from *StockManager*) and the source model unsubstituted parameters (*address* and *date* from *Search*). Figure 8 also emphasizes the ability to apply parameter elements of the source model to parameter elements of the target model. In this case, the first ones (for example *Location*) are substituted by the second ones (*Stock*) in the resulting template.

The *apply* operator must support different construction processes. For the construction of complex systems from a set of parameterized models, it should be possible to elaborate sequences of applications and guarantees consistency properties of the resulting system. We want to exhibit how far it is possible to build the same system model using alternative composition sequences of parameterized models.

For example, Figure 9 shows the design of a complex car hiring system by composing many generic models to the base model shown in Figure 5. This example will serve to illustrate some needs for such sequences. A first need

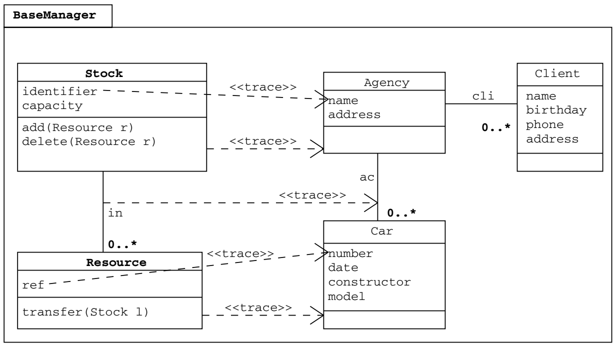


Fig. 7. Base System with Car Management Functionality

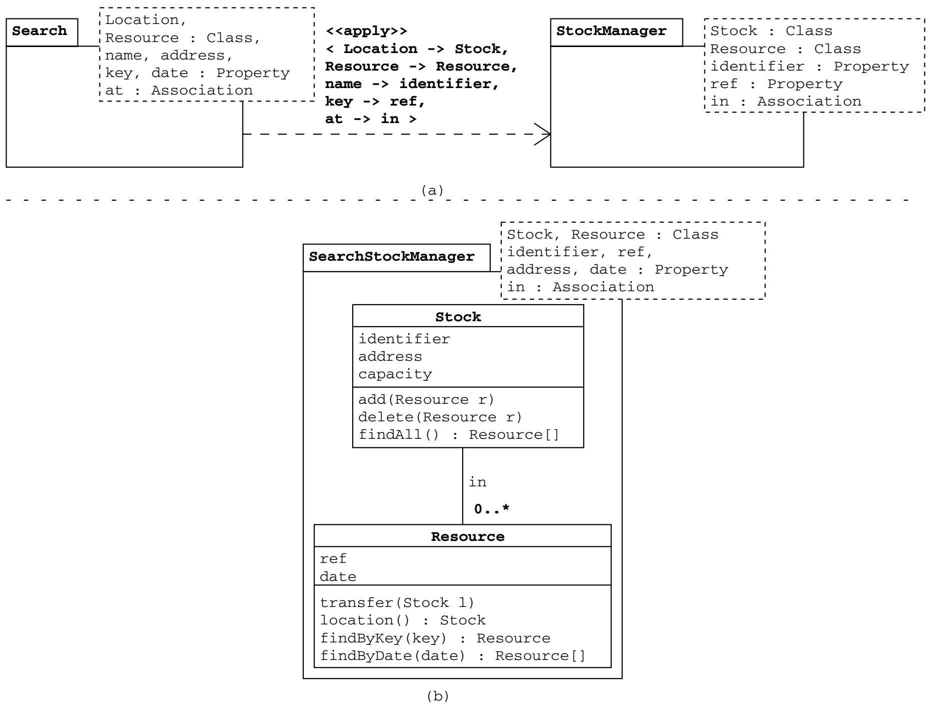


Fig. 8. Template to Template Application

is the ability to apply multiple parameterized model to the same base. When such applications are independent, their evaluation order must not influence the result. It is the case of *Search* and *Allocation* applied to the base.

Another requirement is to express chains of application. Such a chain can be used to apply complex parameterized model resulting from simpler ones. This is illustrated by the application of the *Search* template to the *StockManager* template, explained previously (Figure 8) which produces the *SearchStockManager* model. This new model is then used to add stock management and search functionalities on cars to our system. Note that an alternative construction chain would be to apply the *StockManager* template to the base first, and then the *Search* template to the resulting model. Both chains would produce the same result. We will show this property in the following section. Another example is the application chain *Counting* to *Allocation* to *Base*. Alternative evaluation order of this chain, first *Counting* to *Allocation* then the resulting parameterized model to base or first *Allocation* to *Base* then *Counting* to the enriched *Base*, produce exactly the same result.

In order to provide consistency rules on such building processes, we need a better formalization of parameterized model application. It is the goal of the following section.

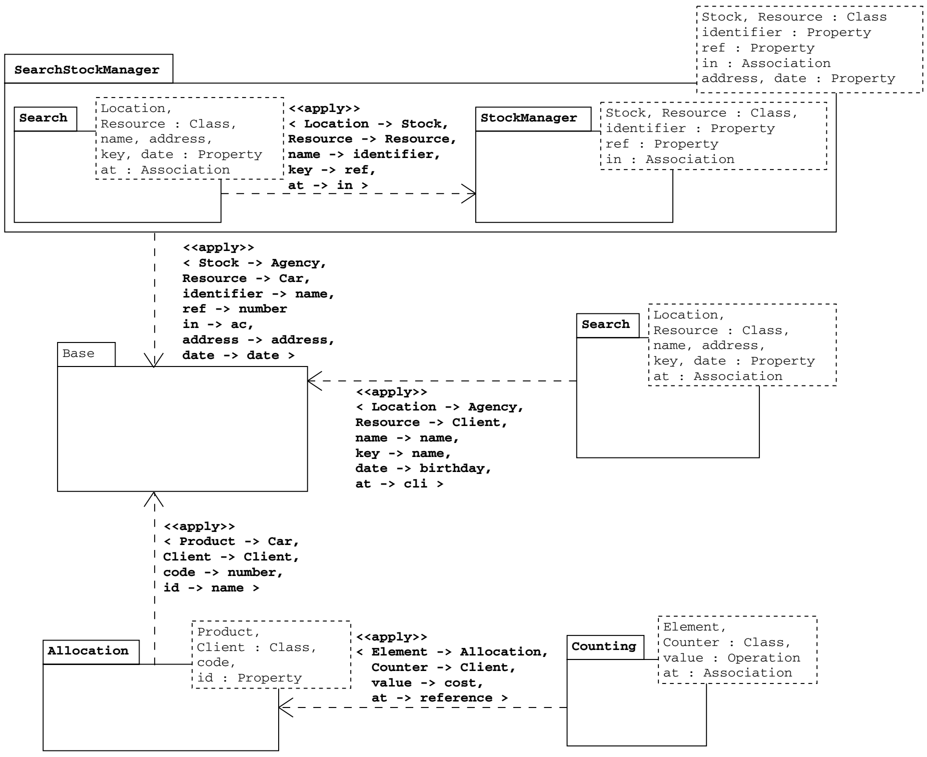


Fig. 9. Car Hiring System

3 Formalization of Parameterized Model Application

The *apply* operator allows to compute a model from the application of a parameterized source one to a target one.

The formalization precises the computation of the resulting model as a set of elements and a set of correspondence relationships between the source model and the target model. These correspondence relationships link elements that represent the same entity[2]. After some definitions, we will state properties of this operator which guarantee the correctness of the previous practices.

3.1 Definitions

Models are considered as sets of model elements that can be classes, attributes, operations or associations. We assume $\mathscr{E}$ to be the set of all these model elements. In case of parameterized models, a model holds a set of parameter elements. As mentioned above, *apply* will construct models that establish correspondence rela-

[2] As the UML2 <<trace>> dependency does.

tionships which are pairs of model elements. The following definition generalizes all these kinds of models.

Definition 1. *A model A is defined by a 3-tuple (E_A, P_A, V_A). E_A is a set of model elements. $P_A \subset E_A$ is a set of parameters. V_A is a set of correspondence relationships defined on $(E_A \times E_A)$.*

Note that in case of models which are not parameterized P_A is empty, and in case of base models V_A is empty.

Definition 2. *Two models are equal if and only if they contain the same set of elements, they own the same set of parameters and they have the same set of correspondence relationships.*

Let two models Z and R, $Z = R \Leftrightarrow \begin{cases} V_Z = V_R \\ E_Z = E_R \\ P_Z = P_R \end{cases}$

Based on these definitions we specify the *apply* operator as follows.

Definition 3. *We write $R = B \underset{s}{\rightarrow} A$ the application of a parameterized model B to a model A according to a substitution set s.*

We note FP_s the set of formal parameters and EP_s the set of effective parameters of s. The resulting model R is constructed according to the following definition rules: $R = B \underset{s}{\rightarrow} A \Rightarrow R = \begin{cases} V_R = V_B \cup s \cup V_A \\ E_R = E_B \cup E_A \\ P_R = (P_B \setminus FP_s) \cup P_A \end{cases}$

Source and target models of a parameterized application cannot share elements: $E_A \cap E_B = \varnothing$.
Formal parameters are elements of the source model: $FP_s \subseteq P_B$.
Effective parameters are elements of the target model: $EP_s \subseteq E_A$.

Note that according to these definitions, parameterized models can be applied to any kind of models, parameterized (see Figure 8) or not (see Figure 6). In the case of parameterized target models, the resulting model is itself parameterized. Recall that resulting parameters are those of the target model plus unsubstituted source model ones. The computation formula of the resulting parameter set (P_R) formalizes this.

3.2 Properties

Assuming these definitions, it is possible to demonstrate a set of properties that guarantees the correctness of application chains and their alternative ordering capacities.

Property 1. *When applying two models to a third one, the order of both applications does not influence the result.*

Let two substitutions set s, s' such as $EP_s \subseteq E_A$ and $EP_{s'} \subseteq E_A$. Then we have
$B \underset{s}{\rightarrow} (C \underset{s'}{\longrightarrow} A) = C \underset{s'}{\longrightarrow} (B \underset{s}{\rightarrow} A)$.

Thanks to this property, it is not necessary to express in which order parameterized models must be applied. This is shown Figure 9. Anyhow *Search* and *Allocation* are applied to the base, the resulting model is the same.

Proof. *Let* $Z = B \underset{s}{\rightarrow} (C \underset{s'}{\longrightarrow} A)$,

$$Z = B \underset{s}{\rightarrow} Z' \text{ with } Z' = (C \underset{s'}{\longrightarrow} A) \Rightarrow Z' = \begin{cases} V_{Z'} = V_C \cup s' \cup V_A \\ E_{Z'} = E_C \cup E_A \\ P_{Z'} = (P_C \setminus FP_{s'}) \cup P_A \end{cases}$$

$$\Rightarrow Z = \begin{cases} V_Z = V_B \cup s \cup V_{Z'} \\ E_Z = E_B \cup E_{Z'} \\ P_Z = (P_B \setminus FP_s) \cup P_{Z'} \end{cases}$$

$$\Rightarrow Z = \begin{cases} V_Z = V_B \cup s \cup V_C \cup s' \cup V_A \\ E_Z = E_B \cup E_C \cup E_A \\ P_Z = (P_B \setminus FP_s) \cup (P_C \setminus FP_{s'}) \cup P_A \end{cases}$$

Let $Y = C \underset{s'}{\longrightarrow} (B \underset{s}{\rightarrow} A)$,

$$Y = C \underset{s'}{\longrightarrow} Y' \text{ with } Y' = (B \underset{s}{\rightarrow} A) \Rightarrow Y' = \begin{cases} V_{Y'} = V_B \cup s \cup V_A \\ E_{Y'} = E_B \cup E_A \\ P_{Y'} = (P_B \setminus FP_s) \cup P_A \end{cases}$$

$$\Rightarrow Y = \begin{cases} V_Y = V_C \cup s' \cup V_{Y'} \\ E_Y = E_C \cup E_{Y'} \\ P_Y = (P_C \setminus FP_{s'}) \cup P_{Y'} \end{cases}$$

$$\Rightarrow Y = \begin{cases} V_Y = V_C \cup s' \cup V_B \cup s \cup V_A \\ E_Y = E_C \cup E_B \cup E_A \\ P_Y = (P_C \setminus FP_{s'}) \cup (P_B \setminus FP_s) \cup P_A \end{cases}$$

As a result, we have $Z = Y \Rightarrow B \underset{s}{\rightarrow} (C \underset{s'}{\longrightarrow} A) = C \underset{s'}{\longrightarrow} (B \underset{s}{\rightarrow} A)$

Property 2. *For any sequence* $B \underset{s_1}{\longrightarrow} (C \underset{s'_1}{\longrightarrow} A)$, *there exists a sequence* $(B \underset{s_2}{\longrightarrow} C) \underset{s'_2}{\longrightarrow} A$, *that produces the same result, such as* $s_2 = s_1 \setminus ((E_A \times \mathscr{E}) \cap s_1)$ *and* $s'_2 = s'_1 \cup ((E_A \times \mathscr{E}) \cap s_1)$.

Figure 10 shows the application of the *StockManager* template to the *Base* then the application of *Search* to the result. This is an alternative construction chain of the application of *Search* to *StockManager* then to the *Base*. This latter construction is equivalent to applying the complex template *SearchStockManager* to the *Base*. We can verify that its parameter sets are conformant to this property.

Parameter elements of *Search* applied to elements of the *Base* in Figure 10 (*address* and *date*) are not concerned by the direct application (Figure 8) of *Search* to *StockManager*. As result, they are transfered to the application of *SearchStockManager* to the *Base* in Figure 9.

In order to prove this property we need the following intermediate property on parameter substitutions:

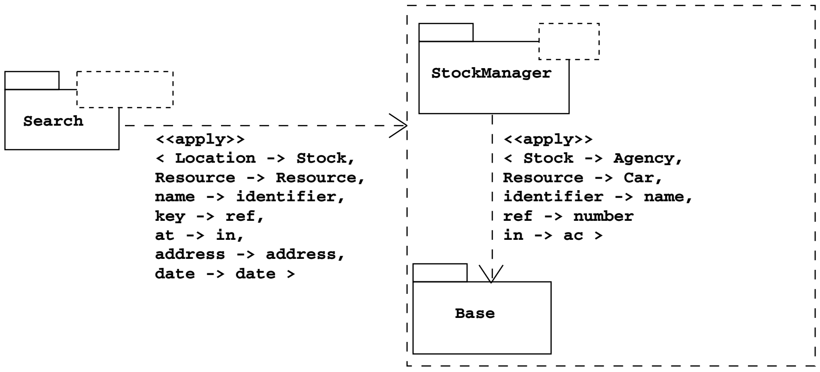


Fig. 10. An alternative to the application of *SearchStockManager*

Property 3. *A parameter can not be substituted more than one time. Once a parameter is substituted, it is no more a parameter in the resulting model.*
$(B \underset{s_1}{\longrightarrow} C) \underset{s_2}{\longrightarrow} A \Rightarrow FP_{s_1} \cap FP_{s_2} = \varnothing.$

Proof. *Let* $(B \underset{s_1}{\longrightarrow} C) \underset{s_2}{\longrightarrow} A = R \underset{s_2}{\longrightarrow} A$ *with* $R = B \underset{s_1}{\longrightarrow} C$.
By definition, $P_R = (P_B \setminus FP_{s_1}) \cup P_C$ *or* $FP_{s_1} \subseteq P_B$ *and* $P_B \cap P_C = \varnothing$ $(E_B \cap E_A = \varnothing$, $P_B \subset E_B$ *and* $P_A \subset E_A)$ *then* $FP_{s_1} \cap P_C = \varnothing$.

$$\begin{aligned}\Rightarrow P_R &= (P_B \setminus FP_{s_1}) \cup (P_C \setminus FP_{s_1}) \\ &= (P_B \cup P_C) \setminus FP_{s_1}\end{aligned}$$

$\Rightarrow P_R \cap FP_{s_1} = \varnothing$ *and by definition* $FP_{s_2} \subseteq P_R$ *so* $FP_{s_1} \cap FP_{s_2} = \varnothing$.

We can now prove the property 2.

Proof. *Let* $R = B \underset{s_1}{\longrightarrow} (C \underset{s_1'}{\longrightarrow} A)$,

$$R = \begin{cases} V_R = V_B \cup s_1 \cup V_C \cup s_1' \cup V_A \\ E_R = E_B \cup E_C \cup E_A \\ P_R = (P_B \setminus FP_{s_1}) \cup (P_C \setminus FP_{s_1'}) \cup P_A \end{cases}$$

According to property 3:
$P_B \cap P_C = \varnothing$ *and* $FP_{s_1} \cap FP_{s_1'} = \varnothing \Rightarrow P_R = (P_B \cup P_C \cup P_A) \setminus (FP_{s_1} \cup FP_{s_1'})$

Let $R' = (B \underset{s_2}{\longrightarrow} C) \underset{s_2'}{\longrightarrow} A$, $R' = \begin{cases} V_{R'} = V_B \cup s_2 \cup V_C \cup s_2' \cup V_A \\ E_{R'} = E_B \cup E_C \cup E_A \\ P_{R'} = ((P_B \setminus FP_{s_2}) \cup P_C) \setminus FP_{s_2'} \cup P_A \end{cases}$

According to property 3:
$P_B \cap P_C = \varnothing$ *and* $FP_{s_2} \cap FP_{s_2'} = \varnothing \Rightarrow P_R = (P_B \cup P_C \cup P_A) \setminus (FP_{s_2} \cup FP_{s_2'})$.

Then we get $R = R'$ *if* $s_1 \cup s'_1 = s_2 \cup s'_2$
Since $s_2 = s_1 \setminus ((E_A \times \mathscr{E}) \cap s_1)$ *and* $s'_2 = s'_1 \cup ((E_A \times \mathscr{E}) \cap s_1)$,

$$\begin{aligned} s_2 \cup s'_2 &= s_1 \setminus ((E_A \times \mathscr{E}) \cap s_1) \cup s'_1 \cup ((E_A \times \mathscr{E}) \cap s_1) \\ &= s_1 \cup s'_1 \cup ((E_A \times \mathscr{E}) \cap s_1) \setminus ((E_A \times \mathscr{E}) \cap s_1) \\ &= s_1 \cup s'_1 \end{aligned}$$

A particular case of property 2 stands when, for each application in the chain, all parameters of the source model are substituted with elements of its target model. In this case, any parameterized model can be directly applied to the next model within the application chain. For such application chains, the evaluation order does not influence the result. This is formalized by the next property.

Property 4. *Let* $B \underset{s}{\rightarrow} C \underset{s'}{\longrightarrow} A$ *an application chain such as* $EP_s \subseteq E_C$, *it can be evaluated either as* $B \underset{s}{\rightarrow} (C \underset{s'}{\longrightarrow} A)$, *or as* $(B \underset{s}{\rightarrow} C) \underset{s'}{\longrightarrow} A$.

Proof. *According to property 2:*
let $B \underset{s}{\rightarrow} (C \underset{s'}{\longrightarrow} A) = (B \underset{s_2}{\longrightarrow} C) \underset{s'_2}{\longrightarrow} A$ *with* $\begin{cases} s_2 = s \setminus ((E_A \times \mathscr{E}) \cap s) \\ s'_2 = s' \cup ((E_A \times \mathscr{E}) \cap s) \end{cases}$

Since $EP_s \subseteq E_C$, $EP_s \cap E_A = \varnothing$ *(by definition* $E_C \cap E_A = \varnothing$*). From this, we deduce that* $(E_A \times \mathscr{E}) \cap s = \varnothing$ *(it can not exist* x *such as* $x \subseteq EP_s$ *and* $x \subseteq E_A$*)*
$\Rightarrow \begin{cases} s_2 = s \\ s'_2 = s' \end{cases}$
Which proves the property.

This last property allows, in Figure 9, to apply *Counting* to *Allocation* to *Base*, without specifying any evaluation order.

4 Related Work

Approaches allowing the decomposition of a system following its functional or technical dimensions aim to simplify the design of information systems. However to form the global system all the dimensions must be assembled. Various approaches exist to express this assembly.

Examples of such decomposition techniques are the Subject-Oriented Design (SOD) approach [8] or the Catalysis approach [10]. The SOD approach proposes the design of an independent model for each concern of the system. These models are called *Subjects* and take the form of a standard UML package. A new type of relation (*CompositionRelationship*) is proposed to compose subjects and express the composition of their elements. A criterion of correspondence can be attached with this relation to indicate whether elements of the same name constituting the composite represents the same entity (default) or not.

The Catalysis approach proposes to decompose the design of systems in horizontal and vertical slices. Vertical slices correspond to a functional decomposition of the system from the points of view of various categories of users. Horizontal slices give a decomposition according to the technical concerns of the system. In this approach, packages are also used to represent the various slices of the system.

These structuring techniques only support the assembly of models designed by analysis of a particular system. They are not suitable for the building of new systems from reusable models [15]. In the following, some approaches have addressed this issue by introducing parameterization techniques.

In [14] we have proposed to consider views as a decomposition technique of systems and their models. A view captures some coherent functional aspect that can be added to a system. Each view is represented as a UML package which contains a model of the functional aspect. The application of such an aspect model to a base model is done by a connection mechanism, which allows to target view model elements to the base ones. The work presented in this paper generalizes this mechanism through a parameterized application process.

The Catalysis approach proposes specific constructions in order to design reusable packages: model frameworks. Those are represented using template packages that are abstract packages containing some elements that must be substituted for being concretized and used. This set of substitutions is defined by a set of element pairs (required element/system element). The Theme approach [9,18] proposes an analysis method of the system (Theme/Doc) which helps in the identification of relations among various functionalities, and a notation (Theme/UML) which allows the formulation of these various functionalities as template packages called *Themes*. A relation (named *bind*) is used to express the parameterized composition of two Themes. In this paper, we focus on parameterized composition chains and their ordering properties. We have focused on structure diagrams. It will be useful to evaluate this formalization on dynamic model elements such as sequence diagrams of Theme or parameterized collaboration diagrams [20].

France *et al.* describe an Aspect Oriented Modelling technique in which aspect and primary models are expressed using UML [17,19]. Aspects are specified by parameterized models. Like SOD, elements of same type and same name are merged to form a single one in the composed model. In order to allow composition of generic aspects (in which elements are named differently as primary model elements) they defined a set of composition directives that can be used to modify the default composition procedure. They also provide directives to state the order of composition between aspects and the primary model. In this paper, we study how far composition orderings are equivalent. These properties are particularly useful in the case of complex composition processes where parameterized models could be composed through alternative composition chains.

Though, our definitions do not impose any targeting strategies, particularly for the structuring of the resulting modelling packages and the realisation of the parameters substitution process. It would be possible, for example, to apply fusion or integration strategies. In this way, the semantic of our apply operator differs from the UML2 merge relationship that defines new elements. It also differs from the MOF combine relationship and the package extension techniques

[7] that impose fusion semantic. Moreover, these relationships are not defined to compose parameterized models.

5 Tool

In the context of MDA approach, we have developed a modelling tool based on the Eclipse Modelling Framework and the UML2 Eclipse plug-in. The Eclipse Modelling Framework is a Java meta-modelling framework that allow to create models in a programmatic way or by a basic (non-graphical) editor. The UML2 Eclipse plug-in is defined by EMF and provides a set of Java classes to handle UML2 models.

Our tool[3] adds a graphical representation and allows defining generic models as UML2 template packages at the PIM level. It provides the functionality of composing these generic models and applying them in order to build a complete system. Strategies to transform composition of parameterized models into platform specific models are offered (see Figure 11).

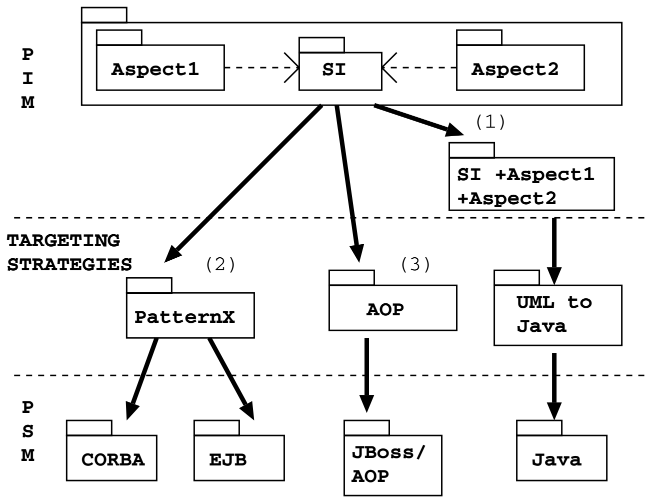


Fig. 11

The simple strategy (1) is to merge all generic models into a single one. Note that this strategy can produce name clashes in the merged model which must be resolved by the user. The resulting model is a standard object-oriented model that can be translated in any object-oriented language (Java in our case). In order to preserve genericity and traceability of our templates down to platform specific models, we have explored two targeting strategies. For the first one (2), we have defined some design patterns allowing to implement generic splited entities and

[3] Available at http://www.lifl.fr/˜mullera/cocoamodeler

experiment them on different platforms [16,6,3]. Our tool can generate IDL and Java code for the CORBA platform according to these patterns. We also explored an AOP targeting strategy (3). We have extended the JBoss/AOP framework to support aspect entities and generate an XML descriptor for their weaving. It is possible to select a targeting strategy specific to each application. The MDA process described in [4] based on marked intermediate PIM is being integrated.

6 Conclusion

Several model driven approaches recognize templates, particularly in the UML sphere, as a powerful technique to specify parameterized models and their usage in the construction of whole system models. We have focused here on parameterized model application which allows to obtain an extended model from an existing one. We have justified concretely and formally some properties which guarantee the correctness of application chains and their alternative ordering capacities.

The formalization is deliberately independent from any specific usage or existing methodologies. It would help in supporting model driven processes and tools dedicated to systems construction by the application and composition of prefabricated generic models. It also allows to combine generic models in order to obtain richer ones.

All this work is integrated into a design tool based on the Eclipse Modeling Framework and the Eclipse UML2 plug-in, from modeling to implementation-level languages. It will give the ability to manage libraries (design, composition, import,...) of parameterized models as standard UML2 templates. This will help in building systems by applying parameterized models.

References

1. OMG Model-Driven Architecture Home Page, http://www.omg.org/mda.
2. Auxiliary Constructs Templates, http://www.omg.org/docs/ptc/03-08-02.pdf, pages 541-568. UML 2.0 Superstructure Specification, 2003.
3. O. Barais, A. Muller, and N. Pessemier. Extension de Fractal pour le support des vues au sein d'une architecture logicielle. In *Objets Composants et Modèles dans l'ingénierie des SI (OCM-SI 04)*, Biarritz, France, jun 2004. http://inforsid2004.univ-pau.fr/AtelierOCMv1.htm.
4. X Blanc, O Caron, A Georgin, and A Muller. Transformation de modèles : d'un modèle abstrait aux modèles ccm et ejb. In *Langages, Modèles, Objets (LMO'04)*, Lille, France, Mars 2004. Hermès Sciences.
5. O. Caron, B. Carré, A. Muller, and G. Vanwormhoudt. Formulation of UML 2 Template Binding in OCL. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings*, volume 3273 of *LNCS*, pages 27–40. Springer, October 2004.

6. Olivier Caron, Bernard Carré, Alexis Muller, and Gilles Vanwormhoudt. Mise en oeuvre d'aspects fonctionnels réutilisables par adaptation. In *Première journée Francophone sur le Développement de Logiciels par Aspects, JFDLPA 2004*, Paris, France, September 2004.
7. A Clark, A Evans, and S Kent. A Metamodel for Package Extension with Renaming. In H Hussmann J-M Jezequel and S Cook, editors, *The Unified Modeling Language 5th International Conference*, Proceedings LNCS 2460, pages 305–320, Dresden, Germany, September 2002.
8. S. Clarke. Extending standard UML with model composition semantics. In *Science of Computer Programming, Elsevier Science*, volume 44, pages 71–100, 2002.
9. Siobhán Clarke and Robert J. Walker. Generic aspect-oriented design with Theme/UML. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 425–458. Addison-Wesley, Boston, 2005.
10. Desmond D'Souza and Alan Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach.* Addison-Wesley, 1999.
11. David S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing.* Wiley, 2003.
12. Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, 2004.
13. S Kent. Model Driven Engineering. In *Proceedings of IFM 2002*, LNCS 2335, pages 286–298. Springer-Verlag, 2002.
14. A. Muller, O. Caron, B. Carré, and G. Vanwormhoudt. Réutilisation d'aspects fonctionnels : des vues aux composants. In *Langages et Modèles à Objets (LMO'03)*, pages 241–255, Vannes, France, January 2003. Hermès Sciences.
15. Alexis Muller. Reusing Functional Aspects : From Composition to Parameterization. In *Aspect-Oriented Modeling Workshop, AOM 2004*, Lisbon, Portugal, October 2004.
16. Olivier Caron, Bernard Carré, Alexis Muller, and Gilles Vanwormhoudt. A Framework for Supporting Views in Component Oriented Information Systems. In *Proceedings of International Conference on Object Oriented Information Systems (OOIS'03)*, volume 2817 of *LNCS*, pages 164–178. Springer, September 2003.
17. Robert France and Geri Georg and Indrakshi Ray. Supporting Multi-Dimensional Separation of Design Concerns. In *AOSD Workshop on AOM: Aspect-Oriented Modeling with UML*, march 2003.
18. Siobhán Clarke and Robert J. Walker. Composition Patterns: An Approach to Designing Reusable Aspects. In *23rd International Conference on Software Engineering (ICSE)*, May 2001.
19. Greg Straw, Geri Georg, Eunjee Song, Sudipto Ghosh, Robert France, and James M. Bieman. Model composition directives. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings*, volume 3273 of *LNCS*, pages 84–97. Springer, 2004.
20. Gerson Sunyé, Alain Le Guennec, and Jean-Marc Jézéquel. Design patterns application in UML. In E. Bertino, editor, *Proceedings of ECOOP 2000*, volume 1850 of *LNCS*, pages 44–62. Springer, 2000.
21. Alan Wills. Frameworks and component-based development. In *Proceedings of International Conference on Object Oriented Information Systems (OOIS'96)*, pages 413–431, 1996.

A Comparative Study of Metamodel Integration and Interoperability in UML and Web Services

Athanasios Staikopoulos and Behzad Bordbar

School of Computer Science, University of Birmingham,
Birmingham, B15 2TT, UK
{B.Bordbar, A.Staikopoulos}@cs.bham.ac.uk

Abstract. The application of MDA to Web services has recently received considerable attention. Similar to UML diagrams, Web services are specialised languages each one targeting a specific aspect and functionality of the system. By using multiple languages, it is possible to specify complete integrated models of the system, having structure, behaviour, communication and coordination mechanisms. To benefit from MDA, Web service languages have to be represented as UML metamodels. In order to provide an overall view of the design and inter-operations of the system with models, it is crucial to integrate their UML metamodels. In this paper, we shall conduct a comparative study of the metamodel integration in Web services and UML. Drawing on the lesson learnt from the integration of Web services, a method of integration of UML metamodels will be presented, which facilitates model transformations and supports interoperability, inter-navigability and consistency across the integrated domains.

1 Introduction

The Model Driven Architecture (MDA) [9] is an emergent technology for software development promoting the automatic creation of models and code, by model transformations. The MDA aims to promote the role of models [7, 15], which are abstractions of the physical system emphasizing particular qualities for a certain purpose and are designed in a UML language or dialect. The model driven approach is based on the metamodel foundation to: a) specify the syntax and semantics of models and b) define the MDA mappings between source and target metamodels [12, 13].

Web services are self-contained, modular software applications that have open, standard based, Internet-oriented interfaces [1]. In a bigger scale, they can be considered as a set of interoperable technologies and standards, designed to support the integration of several autonomous and heterogeneous systems. Web services are particularly geared towards the Service Oriented Architecture (SOA) and paradigm [1, 16], which can be considered as a collection of services, coordinating and communicating with each other to support a specific functionality or concept of a system. A minimalist Web service architecture will contain the SOAP as the communication protocol, the WSDL to describe service interfaces, the UDDI as a service registry and the BPEL to specify executable processes for composite services [5]. If we consider them within an overall architecture, for example SOA, they can specify service-integrated systems, having structure, behaviour, communication and coordination mechanisms.

The application of MDA to Web services is of major importance facilitating their design and development via automation. Recently, designing and developing Web services by MDA have received considerable attention [2, 3, 4, 11]. Despite the fact that Web service models are inherently integrated and created from multiple standards (languages represented as metamodels), their transformations are still considered in isolation and not as part of a whole mechanism. To explain, a business process expressed as a BPEL model has certain dependencies upon service interfaces modelled as WSDL models. This issue should be taken into consideration in the design and transformation of the process. Similar or even more complicated is the case when two processes communicate and exchange data, based on different formalisms such as BPEL and WSCI [1, 5]. Currently, such Web service languages when represented by metamodels appear to be unrelated from each other. In order to obtain a thorough view of the design and the inter-operations of our Web service models, we need to integrate their metamodels.

In this paper, we are examining modelling interoperable Web service systems and architectures, based on different but integrated metamodels. Specifying and formalising their model inter-relations and communication mechanisms is a very important issue for Web services, as they need to collaborate with each other to achieve their common targets. Thus, their capabilities are the result of integration and interoperability. As a result, the integrated metamodelling reflects their real accumulating characteristics and capabilities.

In order, to aid understanding, we need to clarify the terms of metamodel integration and interoperability, used throughout the paper. The descriptions given below are based upon well-established concepts in the field of computing.

- ***Integration***: The creation of links between previously separated computer systems, applications, services or processes.
- ***Interoperability***: The ability to exchange. We need both to a) establish the mechanism to exchange (such as flow of information) and b) define how to extract or understand the information in order to process them.

Finally, the study is broken down into the following sections: Section 2 elucidates the shortcomings of creating isolated metamodels and highlights the advantages of providing integration and interoperability among the UML metamodels. Section 3 investigates the UML and Web service mechanisms, supporting integration, composition and interoperability for their models and schemas respectively. Section 4 compares and analyses them in order to assess their capabilities. Drawing on the lessons learnt, Section 5 presents a method of integrating Web service metamodels, with examples inspired from the Web service domain and their binding mechanism. Section 6 provides an overview of other approaches and various discussion points. Finally, Section 7 presents the conclusions made and summarises the benefits of the approach adopted by the authors.

2 Importance and Benefits

There are cases, where a domain (a subject area, having peculiar set of problems and concepts) [6] such as the Web service domain, may be composed from a number of other sub-domains for example BPEL (process-A), WSCI (process-B), WSDL (description) and SOAP (messaging) sub-domains. As a result, the Web service domain provides the composite or integrated metamodel, describing the overall domain and architecture of a potential system. Obtaining a thorough view of a system, by relating its inter-components is one reason for integrating the metamodels.

Moreover, the (sub) domains have to be interoperable, both horizontally such as processes with processes and vertically such as processes with descriptions and messaging, in order to function and interoperate correctly. Thus, in the first case we have to ensure the definition of meaningful sequence of actions and in the latter to use comprehensive communicative, messaging and transport mechanisms. This is the second reason to equip our metamodels with interoperability mechanisms.

In terms of development, integration and interoperability allows MDA to perform upon a combination of models, with well-established inter-relations and defined interactions. As a result the transformations are equipped with more detailed rules and mappings and generate better-linked artefacts both in terms of specific models and code.

In general, integration allows obtaining a thorough view of the system and its interoperations (assist design) and relating its internal components with established links (provide consistency and formalisation), while interoperability makes our integrated models more operational.

3 Integration and Interoperability Mechanisms in UML and Web Services

Web services have specific mechanisms for supporting the integration and interoperability of their XML artefacts. Similarly, UML has its own specified mechanisms for UML models. As our aim is to apply integration and interoperability among Web service metamodels as precisely as possible and closer to their physical implementation, we have to investigate and compare their mechanisms. In that way, we can apply their original XML integration mechanism to UML Web service metamodels. That will allow us to generate models semantically close to their original characteristics, as the domain metamodels will be equipped with their equivalent XML concepts.

The Object Management Group (OMG) and the World Wide Web Consortium (W3C) are two distinct organisations operating in two different domains, the modelling and specification of software systems and the Web standardisation and interoperability respectively. The OMG uses the UML family of languages and standards for software design and development, which are based on graphical models. UML languages are specified and defined upon a common core language, the Meta Object Facility (MOF) [12] providing the fundamental building blocks to construct and store metamodels. From the other side, the W3C uses the XML family of languages that are textual specifications to specify Web services and standards. The XML languages are defined upon the XML Schema Definition (XSD) [19] to describe and constraint the structure and content of XML documents.

Both specification mechanisms (MOF in UML and XSD in XML) are comparable and can been seen as meta-languages, languages to define and create other languages. Thus, XSD is used to define WSDL and BPEL, and MOF to define UML and CWM [12]. As they are created from a common metamodel (XSD), the service languages conform to and share common semantics, so it is much easier to be integrated and be interoperable. Meta-languages play a very important role, as they define core domain characteristics that can be reused to define and create a variety of other metamodels, belonging to the same family of languages [13].

The UML specification is defined using the metamodelling approach and mechanism. The typical role of metamodelling is to define the semantics of how the model elements in a model get defined and instantiated. Web services on the other hand are defined by using a very flexible mechanism, the XML Schema, providing a way to specify the structure of XML documents.

Finally, the UML specification is organised into Infrastructure and Superstructure architectures. As a result, the various constructs defined are highly reused and coupled within the overall UML architecture. On the other hand, Web services can be seen as more independent entities that are part of more flexible or less coupled architectures. They can be integrated and combined in various ways with the flexibility of a component. An example of such architecture is the SOA [16], which is organised in separate layers regarding the functionality and concepts provided. In that sense the architecture can be seen as an accumulation of aspects.

Following, we identify, describe and compare the mechanisms defined within the UML and Web Service specifications supporting the fundamental ideas of composition, inter-relation, communication and extensibility of meta- elements.

3.1 Mechanisms for the Integration of UML Metamodels

The UML specification is defined in a number of metamodels and organised within hierarchical packages. The metamodels are gradually built upon more abstract modelling concepts, defined within fundamental reusable packages, such as the Infrastructure Library [13] and the Superstructure Kernel [14]. The UML language is organised in a four-layer metamodel architecture, separating the instantiation concerns across different layers, for example MOF from UML. According to that principle, the instantiation of meta-classes is carried out through MOF. The UML architecture has been designed to satisfy the following criteria [13].

1. **Modularity:** Group constructs into packages by providing strong cohesion and loose coupling.
2. **Layering:** Support a package structure to separate meta-language constructs and separate concerns (regarding instantiation).
3. **Partitioning:** Organise conceptual areas within the same layer.
4. **Extensibility:** Can be extended in various ways.
5. **Reuse:** UML metamodel elements are based upon a flexible metamodel library that is been reused.

To realise the above qualities, the UML specification is organised into two parts: Firstly, the UML Infrastructure defining the foundation language constructs where both M2 (UML) and M3 (MOF) meta-levels of the four-layer metamodel architecture are being reused. Secondly, the UML Superstructure extends and customises the Infrastructure to define additional and more specialised elements making up the modelling notions of UML.

The entire UML specification can provide an example of how metamodels representing different concepts, such as structure by a class or a component diagram and behaviour by an activity or an interaction diagram are integrated and connected together. In that case, even if they are defined within different packages they are interrelated and share common basic elements with other metamodels.

3.1.1 The UML Infrastructure

The Infrastructure Library [13] provides the basic concepts for organising and reusing metamodel elements. In the case of metamodel relation and composition, these are as follows:

Model elements to group elements together such as a) *Visibilities,* provide basic constructs for visibility semantics b) *Namespace,* provides concepts for defining and identifying a model element within a namespace c) *Package,* groups elements and provides a namespace for the grouped elements.

Model elements to specify some kind of relationship between elements such as a) *Generalisation,* specifies a taxonomic relationship between a more general and a more specific classifier b) *Redefinition,* specifies a general capability of redefining a model element c) *Association,* both a relationship and a classifier. Specifies a semantic relationship between typed instances d) *Element Import,* a relationship that identifies an element in another package, allowing the element to be referenced using its name without a qualifier.

Model elements defining basic mechanisms for merging their content such as a) *Package Merge,* specifies how one package extends another package, by merging their contents through specialisation and redefinition and b) *PackageImport,* a relationship that allows the use of unqualified names to refer to package members from other namespaces.

3.1.2 The UML Superstructure

The UML Superstructure [14] relies on the essential concepts defined in the Infrastructure to build the UML 2.0 diagrams. It provides a clear example of metamodel integration, as it brings together different packages from the Infrastructure library, by using package imports and merges. It redefines some of the concepts and further extends their capabilities [13].

The Superstructure in order to support the concepts of integration and interoperability among metamodels provides *additional relationships* [14] and links among elements. Examples of such links and relationships are a) *Dependency*, a relationship that signifies that a model element requires other model elements for their specification or implementation b) *Abstraction,* a relationship that relates two elements or sets of elements, representing the same concept at different levels of abstraction c) *Usage,* when one element requires another element (or a set) for its full implementation or operation d) *Permission,* signifies granting of access rights from the supplier to a client model element e) *Realisation,* a specialised relationship between two sets of model elements, one specifies the source (supplier) and the other implements the target (client) f) *Substitution,* a relationship between two classifiers, implying that instances can be at runtime substitutable, where instances of the contract classifier are expected g) *Implementation,* a relationship between a classifier and an interface signifying that the realising classifier conforms to the contract specified by the interface.

In addition, it defines a number of *composite structures* [14], providing more complicated elements with advanced capabilities. Examples of such elements are a) *Components*, representing a modular part of a system, which is replaceable within its environment b) *Connector*, a link enabling communication between instances. It can be something as simple as a pointer or something as complex as a network connection c) *Composite Structures*, a composition of interconnected elements, representing runtime instances collaborating over communications links to achieve some common objectives d) *Ports*, a structural feature of a classifier specifying a distinct interaction point between that classifier and its environment e) *InvocationAction*, invoke behavioural features on ports from where the invocation requests are routed onwards on links, deriving from attached connectors f) *Collaborations*, a kind of classifier defining a set of cooperating entities to be played by instances (its roles) and a set of connectors defining communication paths between the participating instances g) *CommunicationPath*, an association between two nodes, enabling them to exchange signals and messages.

3.1.3 Extensibility – Provide User Defined Elements

The UML specification is flexible supporting two types of extensions [7, 9, 12]. The first one is based on profiles and is referred to as a lightweight built-in mechanism. It does not allow the modification of existing metamodels but rather their adaptation with constructs that are specific to a particular domain, platform, or method. The second approach uses metamodelling techniques by explicitly defining new metamodel elements from pre-existing metamodels like the ones defined in MOF or the UML Infrastructure. This approach allows ultimate extensibility as it enables the definition of new concepts with new capabilities that can be tailored to represent precisely a particular domain of interest. There are various examples of extending UML, either with metamodels or profiles such as EDOC and the EJB Profile respectively.

3.2 Mechanisms for the Integration of Web Service Standards

The Web service standards are built-upon the XML and XML Schema. As a result many of their characteristics and capabilities, such as extensibility and referencing, are based upon their defined concepts. By design, almost all Web service standards (as loosely coupled) [1] are designed to accommodate their integration and interoperability aspects flexibly. So, they define various points of extensibility, allowing them to interoperate with other standards and usually are separated into *abstract* and *concrete* parts.

The Web service implementations rely on the collaboration of various specifications to make them really functional and interoperable. For example, the SOA [16] is based on a collaboration of various Web service specifications, such as service description, discovery and messaging. Each one can represent either a particular layer (regarding the SOA architecture) or a particular functionality or concept. In this sense, they can be compared with UML modelling that is based on a combination of various metamodels, belonging to different packages, to make up a complete system specification.

An example of how Web service specifications can be combined and interoperated to fulfil an objective is a business process request defined as a BPEL *Invoke* operation [5]. The process defines the implementation logic of a service, accessed from specific interaction points via a set of Web interfaces defined in WSDL and sending a SOAP message representing a particular request to a business participant [1]. The participant replies back by triggering an equivalent mechanism, with an appropriate formatted XML message.

In this paper, we are interested in investigating these inter-relationships and dependencies among fundamental Web service standards such as BPEL, WSDL and SOAP and examine how they are combined to support service interoperability at different levels of abstraction and implementation, by realistic examples and cases. The following are defined elements and technologies used for the integration, extensibility, collaboration and communication of schemas and Web service specifications.

3.2.1 Common – Core Characteristics

The XSD mechanism [19] provides the core/fundamental characteristics of Web services, utilised by almost all Web service standards, allowing mechanisms for grouping, extensions and referencing. The XSD also provides the foundation mechanism for constructing messages and datatypes that are the means of interoperability and communication for services.

The elements supporting such mechanism are a) *Schema*, is associated with a namespace and provides a grouping for defined XML elements b) *Namespace*, provides identification and access rights for its elements c) *Import*, allows to use schema components across different namespaces with references d) *Include,* assembles schema components to a single target namespace from multiple schema definitions e) *Redefine,* allows the redefinition of one or more components f) *Redefinable,* specifies an element so that it can be redefined g) *SubstitutionGroup,* supports the substitution of one named element for another h) *Extension,* provides the mechanism to extend the element content i) *Restriction,* provides the mechanism to restrict the element content j) *References,* provides pointers to already defined elements such as *GroupRef, AttributeGroupRef* and *ElementRef.* Finally, k) *AnyURI*, *AnyType and AnyAttribute* can point to any location, any type and extend an element with attributes not specified by the schema respectively.

3.2.2 Specialised Characteristics

More specialised characteristics and concepts are usually defined within each service specification. For example, the WSDL defines how Web service interfaces can be defined in terms of protocol bindings, port-types, operations and messages. Analogously, other specific application constructs have been defined for BPEL and SOAP [1, 11]. Those standards can be combined and integrated together by specifying a binding mechanism that actually provides the links among the involved service specifications. So, tools and engines can realise those implementations and execute the actual Web service collaborations across different standards and protocols.

Almost all Web service standards are designed to accommodate interoperability and be extensible. As a result, their specifications at certain points, particularly their concrete parts are defined in a way that leaves space for different implementations. It is very similar to UML or Java when a *Classifier* or an *Object* can be of any type, therefore the model or application can support different implementations or behaviours.

3.2.3 Auxiliary Characteristics

There are also various auxiliary technologies and standards that are used to provide more sophisticated mechanisms, such as for relating elements across multi-documents. Those can be easily used and embedded within specifications to create more elaborate structures and complicated functionalities. Such a mechanism is the *XPath* [18] language for addressing various parts of an XML document. It provides the means of linking elements together; therefore it can be represented by a relationship in UML modelling.

4 Analysis and Classification of Mechanisms

At this point, we should analyse and classify the integration and interoperability mechanisms supported by UML and Web services, upon the following fundamental criteria:

Structures: Provide the ability to implement a) *Containers*, group and identify model elements within collections and b) *Composite Structures*, provide the means and concepts to integrate/combine together different model elements into new or coupled entities.

Dynamics: *Messaging*, establish the concepts and mechanisms to specify dynamics for example perform invocations, interactions, messaging upon established connections.

Links: Provide *Relationships/Addressing* mechanisms to establish semantic relationships among model elements that may belong to different groups.

Mechanisms: Define *Mechanisms upon containers,* operate upon elements or collections of elements. New groups of elements may be generated as a result of intersection and union operations.

According to these criteria, the UML and Web service capabilities are compared, assessed and described as in the following tables:

Table 1. The UML mechanism supports criteria by specifying the following UML modelling elements

	UML support
Containers	Initially provided by the Infrastructure library and specialised from the Superstructure kernel. Examples are Namespace, Visibility and Package model elements
Comp. Structures	Various metamodel elements are defined such as components, composite structures, collaborations
Messaging	A number of actions are specified for messaging, invocations and their supporting concepts like input and output pins
Links	There are various types of semantic relationships such as permissions, substitution, generalisation, usage etc
Mechanisms	Package mechanisms, like package import and merge

Table 2. The Web service mechanism supports criteria by specifying the following XML elements

	Web service support
Containers	Schema, Namespace
Comp. Structures	ComplexType, Group, Sequence etc
Messaging	XML Datatypes, SOAP messages
Links	Element, Attribute and Group references, Ids
Mechanisms	Import, Include, Redefine

4.1 Compare and Contrast Mechanisms

It is clear that both XSD and MOF or the UML Infrastructure are meta-languages for the modelling and Web service domain respectively. As such, they provide the fundamental concepts and building blocks to define and create other languages such as the WSDL, the BPEL or the UML and the CWM. Regarding the four-layered metamodel architecture they both belong to M3 level [12]. In addition, they are both self describing and reflective as they have been specified by their own means and concepts.

The UML and Web service specifications, as seen in section 4, both define integration mechanisms, however they are designed in such a way to support and reflect the characteristics of their domains. Therefore, UML is designed to create more compact models through meta-models. UML also seems more integrated as it is designed from one organisation all-over. Metamodels via diagrams can provide different views of the same system. In that case, their integration points are well defined and bound, as effectively they belong to the same specification. On the other hand, Web service standards are more loosely coupled and are developed rather independently from each other. They are integrated and combined later on into functional units, using previously defined, abstract integration points. In addition, their specifications are defined in textual XML Schemas.

To conclude, we can say that the UML specification is focused on separating the different views of a system by providing different diagrams that are somehow related together by their metamodels, while Web service specifications are concerned with logically separating the *abstract* from the *concrete* parts [1, 19].

The result of the comparison will justify and influence the approach adapted, of how to implement the integration and interoperability mechanism with its domain metamodel, in a way to reflect as accurately as possible their actual characteristics.

5 A Method to Support Metamodel Integration and Interoperability

Currently, the Web service domain is scattered into several different metamodels such as BPEL, WSDL and SOAP [1]. Our approach on supporting the integration and interoperability of metamodels is based on the concept of relating all these unconnected metamodels in a cooperating fashion, through specified and formalised links. As a result, we introduce the binding mechanism (please refer to Fig. 1) as a metamodel. The method is influenced from the Web service domain. However, we believe that the approach is similarly applicable to other domains, as it can be considered neutral and generic enough. The binding metamodel integrates the metamodels together in a formalised way, by specifying both their static aspects (by defining interrelationships) in order to provide integration and dynamic aspects such as communication, interactions and conformance to provide interoperability among the metamodels.

The steps to apply our approach can be described as follows:

Firstly, we need to identify the metamodels composing our system or domain of interest. One can either create these metamodels or reuse predefined ones. Secondly, the relationships, dependencies or interaction points among these metamodels have to be identified. Thirdly, the mechanisms supporting integration and interoperability between the actual *"real"* domain (in the case of Web services, by XML and XSD) and modelling domain (by metamodels) are compared and checked whether they are harmonised. This should allow to design their actual supported mechanisms (originally expressed in XSD) in another formalism (metamodel) as precisely as possible, providing clear domain models and encapsulating their original domain capabilities accurately. Having identified the supported mechanisms (for example how to relate to elements), the binding metamodel is introduced, where the relationships, properties and rules are applied accordingly. At this stage the integration points among the metamodels are being defined. Following, upon the integration points (providing relationships and links) and within the binding metamodel, we further specify the communication mechanism in terms of interactions, messages and data-types exchanged, making our models functional and interoperable. Lastly, we may have to model the binding mechanism with equivalent UML models as precisely as possible in order to respect its internal semantics for example establish object roles within collaboration diagrams.

5.1 Applying the Binding Mechanism – Web Service Examples

Following, a number of examples from the domain of Web services are provided to illustrate how the binding metamodel can be applied among Web service metamodels. As already mentioned, the Web service domain can be represented by various metamodels. In these examples, we are going to consider two metamodels: the BPEL and WSDL, representing a partial view of the Web service domain. The metamodels can either be created from their original XSD specifications or reuse pre-existing ones [2, 3, 4].

Fig. 1 illustrates how the Web service domain is represented by the BPEL and WSDL metamodels and how these are mapped to equivalent UML modelling concepts, providing their platform independent representations, as UML activity and component diagrams respectively. Regarding these mappings, there are already some research activities defining their MDA transformations [2, 4]. On the right side, one can see how a metamodel binding between the BPEL and the WSDL metamodels is applied and how this is reflected as a UML representation (in this case a collaboration element). The UML model that would be selected needs to encapsulate the binding mechanism as semantically precise as possible and provide all the necessary concepts and capabilities for representing integrations and collaborations of its participating parts.

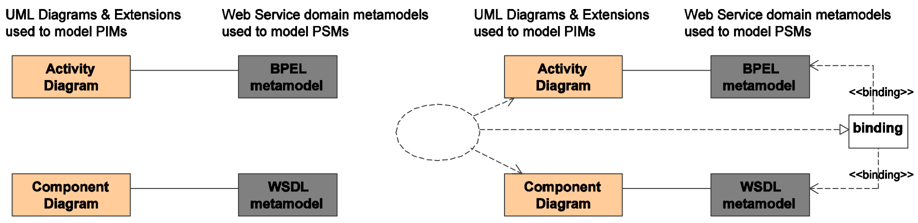

UML Diagrams & Extensions used to model PIMs
Web Service domain metamodels used to model PSMs
Activity Diagram
BPEL metamodel
Component Diagram
WSDL metamodel
UML Diagrams & Extensions used to model PIMs
Web Service domain metamodels used to model PSMs
Activity Diagram
BPEL metamodel
<<binding>>
binding
<<binding>>
Component Diagram
WSDL metamodel

Fig. 1. The UML & Web service domains with equivalent mappings (on the left) and required bindings (on the right)

The binding metamodel in Fig. 2, represents the BPEL and WSDL integration. It defines two additional model elements, the *PartnerLinkType* and *Role* together with their inter-relationships with the other model elements from BPEL and WSDL. In this case, the binding metamodel is attached to the actual WSDL metamodel as an extension, meaning that both WSDL and binding model elements share the same namespace.

The services with which a business process interacts are modelled as *PartnerLinks* and are performed upon Web service interfaces. Each *PartnerLink* is characterised by a *PartnerLinkType* maintaining the conversational relationship between two services by defining *Roles* played by each of the services in the conversation 19. Each *Role* specifies exactly one WSDL port type. The relationship among partners is typically peer-to-peer (such as BPEL to BPEL) and can be modelled by a two-way dependency. The *PartnerLink* declaration specifies the static shape of the relationship that the process will employ in its behaviour.

Afterwards, the interaction mechanism of the binding metamodel is being specified, by identifying its dynamics in terms of establishing interactions, message exchanges in various patterns and use comprehensive data-types as illustrated in Fig. 3. That will permit our models to be interoperable across their integrated points, meaning that their instances can be really executable [17]. The integrated points defined as relationships among model elements specify the links where messages or signals can travel. Such formalisation provides consistence, as interactions can be performed only through these established points following a specific interaction pattern.

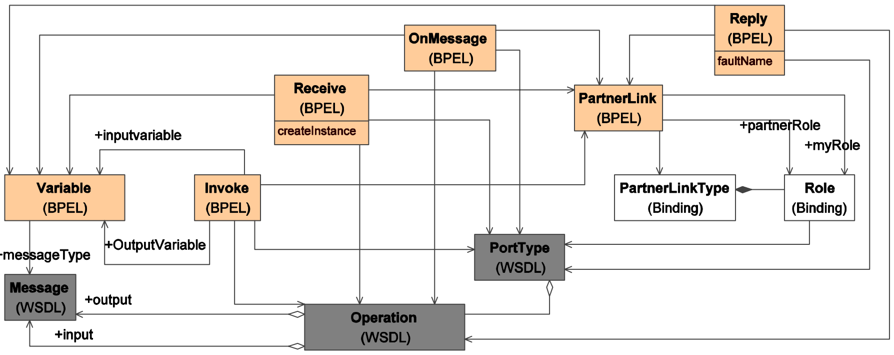


Fig. 2. A metamodel binding example between BPEL & WSDL supporting integration

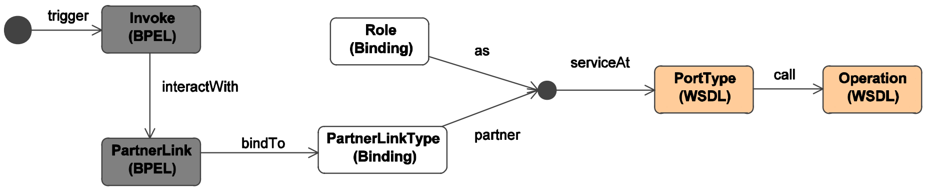
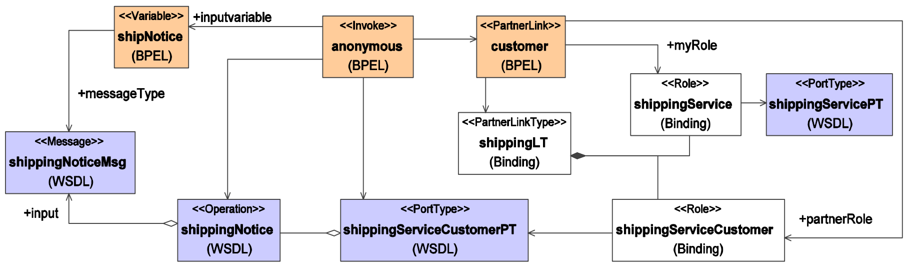


Fig. 3. How the metamodel binding supports interoperability

In particular, Web services by design support interoperability, as they are based on XSD for specifying datatypes, SOAP for messaging and communications and WSDL for describing interactions as exposed operations. Thus, a Web service example as in this case would not encounter severe inconsistency problems of that kind, as the interactions would be performed upon common standards. For example, in order to perform a simple invocation call across collaboration processes, the following sequence will occur among involved metamodel elements to transmit a message across the integrated points and according to a specific interaction pattern (in this case without a reply):

$$\text{BPEL1} \rightarrow \text{binding1} \rightarrow \text{WSDL1} \rightarrow \text{SOAP} \rightarrow \text{WSDL2} \rightarrow \text{binding2} \rightarrow \text{BPEL2} \qquad (1)$$



<<Variable>> shipNotice (BPEL)
+inputvariable
<<Invoke>> anonymous (BPEL)
<<PartnerLink>> customer (BPEL)
+myRole
+messageType
<<Role>> shippingService (Binding)
<<PortType>> shippingServicePT (WSDL)
<<PartnerLinkType>> shippingLT (Binding)
<<Message>> shippingNoticeMsg (WSDL)
+input
<<Operation>> shippingNotice (WSDL)
<<PortType>> shippingServiceCustomerPT (WSDL)
<<Role>> shippingServiceCustomer (Binding)
+partnerRole

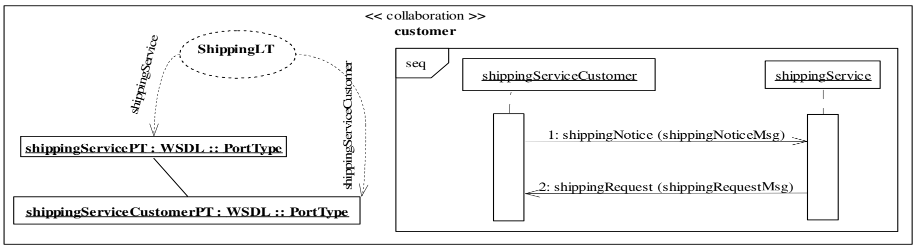
Fig. 4. Instance Examples upon binding

Following, Fig. 4 illustrates how the metamodel mechanism can be realised by a set of instances, participating in an invocation operation *Invoke* from a *shippingServiceCustomer* to another *shippingService* participant in order to handle the shipment of orders. The actual example is derived from the BPEL specification [5].

Finally, we examine how to represent the binding mechanism in a platform independent manner with an equivalent UML model. For that reason, one needs to utilise the UML 2.0 composite structures, such as *Collaboration* and *StructuredClasses* that can support the composition of interconnected elements, representing run time instances, collaborating over communication links to achieve common objectives [14]. In particular, *Collaborations* provide the means to define common interactions between objects and other classifiers and assign responsibilities in terms of roles. The interaction is similar to providing a pattern of communication between parts and is specified as a set of messages passed between objects playing predefined roles.

In this case, the UML 2.0 Collaboration modelling element is used to encapsulate the *PartnerLinks* playing various *Roles* by each of the services participating in the conversation and providing a semantically mapping to our domain binding metamodel.

Following, Fig. 5 depicts the binding mechanism as an independent model, explaining how the binding mechanism works, by describing the structure of collaborating elements (roles), performing specialised functions upon defined communication paths of participating instances. In that case the binding metamodel and mechanism are rather simple and can be represented by a UML *AssociationClass* [13]. That is to say a class that defines specific properties upon an established semantic relationship between classifiers, in this case the metamodel elements from BPEL and WSDL.



<< collaboration >>
customer
ShippingLT
shippingService
shippingServiceCustomer
shippingServicePT : WSDL :: PortType
shippingServiceCustomerPT : WSDL :: PortType
seq
shippingServiceCustomer
shippingService
1: shippingNotice (shippingNoticeMsg)
2: shippingRequest (shippingRequestMsg)

Fig. 5. UML collaboration, modelling bindings in UML

6 Related Work and Discussion

There are several approaches on metamodel composition and interaction [8, 10, 15]. More specifically, in [10] the key feature of this approach is the combination of new metamodels from existing metamodels, through the use of newly defined operators such as equivalence, implementation and interface inheritance. The approach emphasises the compositional operations for creating new concrete metamodels and not actually relating them as cooperating entities, as in our case. Next, the approach in [15] suggests the integration of metamodels by a joint action model. The approach is based on message interactions, thus supporting more dynamically and interoperable models. In that respect, the approach is comparable to our method on establishing interaction points across the metamodels. Finally, the approach in [8] is based on extending the UML language for the composition of domain metamodels by proposing a UML profile.

In our study our effort is to establish both connections upon the metamodel elements to support integrations in a cooperating way and communication mechanisms to support their interaction characteristics. In that way we can provide consistency and formalisation upon the interconnected metamodels, properties that are necessary for performing model transformations.

7 Conclusion

Integration and interoperability are very important issues for composite domains as in the case of Web services, as they need to collaborate with each other to achieve their common targets. Their emerged capabilities can be seen as the product of integration and interoperability. In modelling, Web service languages need to be represented by equivalent metamodels. By defining their metamodel relationships and interaction mechanisms, we support the integration and interoperability of their models. In that way we can design complete system models having formalised interaction points and perform model transformations across multiple connected metamodels.

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, V., Web Services Concepts, Architectures and Applications, Data-Centric Systems and Applications, ISBN: 3-540-44008-9, (2004)
2. Bézivin, J., Hammoudi, S., Lopes, D., Jouault, F.: An Experiment in Mapping Web Services to Implementation Platforms. Technical report: 04.01, LINA, University of Nantes, Nantes, France (2004).
3. Bordbad, B., Staikopoulos, A.: Modeling and Transforming the Behavioural Aspects of Web Services. In: Proc. 3rd Workshop in Software Model Engineering - WiSME2004, UML (2004)
4. Bordbar, B., Staikopoulos, A.: On Behavioural Model Transformation in Web Services. In: Proc. Conceptual Modelling for Advanced Application Domain (eCOMO), Shanghai, China (2004), p. 667-678
5. BPEL: BEA, Microsoft, IBM, SAP, Siebel, Business Process Execution Language for Web Services, Version 1.1. (2003)

6. Greenfield, J, Keith Short: Software Factories, Wiley, ISBN: 0471202843, 2004
7. J. Siegel, Developing in OMG's Model Driven Architecture, Object Management Group, November (2002)
8. Jacky Estublier, A.D.I. Extending UML for Model Composition. in Australian Software Engineering Conference (ASWEC). 29 March, 1 April (2005). Brisbane, Australia.
9. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture-Practice and Promise. (2003)
10. Ledeczi A., G.N., G. Karsai, P. Volgyesi, M. Maroti. On Metamodel Composition. in IEEE CCA 2001. September 5, (2001). Mexico City, Mexico.
11. Lopes, D., Hammoudi, S.: Web Services in the Context of MDA. In: Proc. 2003 International Conference on Web Services (ICWS'03) (2003)
12. OMG: Meta Object Facility 2.0 Core Specification. (2003). Document id: ptc/03-10-04
13. OMG: UML 2.0 Infrastructure Specification. Document id: ptc/03-09-15 (2003)
14. OMG: UML 2.0 Superstructure Specification. Document id: ptc/03-08-02 (2003)
15. P. Denno, M.P.S., D. Libes, E.J. Barkmeyer, Model-Driven Integration Using Existing Models. IEEE Software, Sept./Oct. (2003): p. 59-63.
16. Rakesh Radhakrishnan, Mike Wookey, Model Driven Architecture Enabling Service Oriented Architectures, Sun Micro Systems, March (2004)
17. Stephen J. Mellor and Marc J. Balcer: Executable UML a Foundation for Model Driven Architecture. Addison Wesley. ISBN 0-201-74804-5, 2002
18. W3C, XML Path Language (XPath) 2.0, W3C Working Draft. July (2004)
19. XML Schema W3C, XML Schema Part 0: Primer, W3C Recommendation, May (2001)

Control Flow Analysis of UML 2.0 Sequence Diagrams

Vahid Garousi, Lionel C. Briand, and Yvan Labiche

Software Quality Engineering Laboratory (SQUALL),
Department of Systems and Computer Engineering, Carleton University,
1125 Colonel By Drive, Ottawa, ON K1S5B6, Canada
{vahid, briand, labiche}@sce.carleton.ca

Abstract. This article presents a control flow analysis methodology based on UML 2.0 sequence diagrams (SD). In contrast to the conventional code-based control flow analysis techniques, this technique can be used earlier in software development life cycle, when the UML design model of a system becomes available. Among many applications, this technique can be used in SD-based test techniques, model comprehension and model execution in the context of MDA. Based on the well-defined UML 2.0 activity diagrams, we propose an extended activity diagram metamodel, referred to as Concurrent Control Flow Graph (CCFG), to support control flow analysis of UML 2.0 sequence diagrams. Our strategy in this article is to define an OCL-based mapping in a formal and verifiable form as consistency rules between a SD and a CCFG, so as to ensure the completeness of the rules and the CCFG metamodel with respect to our control flow analysis purpose and enable their verification. Completeness here means if the CCFG metamodel has all classes and associations needed, and the rules are adequate with respect to our purpose. Furthermore, we define Concurrent Control Flow Paths, which are a generalization of the conventional Control Flow Path concept. The control flow analysis technique is applied to an example SD to demonstrate the feasibility of the approach.

1 Introduction

Control Flow Analysis (CFA) [1] is a widely used approach for analyzing programs. A Control Flow Graph (CFG) [1] represents all alternatives of control flow in a program. The concept of CFG was first used as a data structure in compilers [1] for code optimization purposes. Later, it was adopted extensively in the software engineering and particularly in the software testing community (e.g. [2, 3]). In a CFG, different paths from the start node to the end node are called Control Flow Paths (CFP), which show the different paths a program may follow during execution. Furthermore, Data Flow Analysis (DFA) and data flow-based testing techniques (such as [4, 5]) are also, to a great extent, based on CFA: different CFPs, derived by CFA, are examined to extract data flow information such as *def-use* pairs (definition-uses of data).

Based on the source of information to derive the control flow information of a system, we can divide the CFA techniques into two groups: *code-based* and *model-based*. Code-based CFA (*CBCFA*) is the traditional CFA [1], in which control flow information is derived from the available source code. CBCFA has been greatly used in the software testing literature [2, 3, 6, 7] and owns strong mathematical foundations [1].

On the other hand, we define model-based CFA (*MBCFA*) to be the derivation of control flow information from the design model (such as UML [8-10]). To the best knowledge of the authors, there have been few works [11-15] on MBCFA and in particular based on UML. UML provides ways to model the behavior of an Object-Oriented (OO) system using interaction (sequence and collaboration) diagrams. However, as it will be discussed in Section 2.1, analysis and derivation of control flow information are not straight-forward in UML interaction diagrams, and most specifically sequence diagrams (SD). There are challenges such as impact of asynchronous messages and intra-SD parallelism due to *par* interaction operators.

The motivations for our work are twofold: (1) When and where is MBCFA preferable over CBCFA? and (2) What is missing in the existing works and has to be addressed by a comprehensive MBCFA technique? Some of the advantages of MBCFA over CBCFA are easier extraction of concurrent control flow and dynamic control flow information (such as polymorphism), which will benefit applications of CFA such as testing and refactoring. As another benefit, MBCFA can be used in model-based test techniques, earlier in the software development life cycle, i.e., as early as the UML design model of a system is available. Our MBCFA technique can be useful in most of the existing SD-based testing techniques, such as [16-18]. Some of the possible applications of our approach in the context of MDA [19] include the testing phase of the MDA software development life cycle (Figure 1-2 of [19]) and also model compilation, comprehension, and execution. Model execution, model comprehension, model checking, model optimization, and model walk-through are other possible applications of MBCFA in the context of MDA. These tentative application areas remain to be further investigated.

1.1 Related Works

To the best knowledge of the authors, there have been few MBCFA techniques [11-15] based on UML. These strategies are reported in Table 1 and are compared according to eleven criteria: (1) The source model on which a MBCFA technique is applied. It can be either SD or reverse-engineered SD (RE-SD); (2) UML version supported: UML 1.x (1.3, 1.5) or 2.0; (3) The output Control Flow Model (CFM) produced by a MBCFA technique; (4) The degree of formality employed when defining the CFM (if any), ranging from a set of examples to a formal representation (such as a metamodel); (5) The degree of formality employed in defining the transformation/mapping from SDs to the CFM (if any), ranging from a set of examples to formal mapping rules; (6) Whether the MBCFA technique supports the control flow resulted from asynchronous messages in SDs; (7) Whether inter-SD control flow is supported, i.e. when SDs refer to each other using *ref* construct in SDs; (8) If loops are supported; (9) Whether conditions are supported; (10) Support for polymorphism in SD lifelines. When an operation in a parent class is involved in a SD, overwritten versions of this operation have to be accounted for during CFA as varying control flows may occur at runtime; and (11) If the technique provides a formalism for CFPs and offers an algorithm to derive CFPs from a CFM.

As Table 1 shows, all of the existing works are based on UML 1.x versions. Only one of them proposes a formal syntax for the produced CFM [11]. Only two ([13] and [14]) have formal transformation/mapping techniques from SDs to their chosen CFM. None of the existing works support inter-SD control flow and polymorphic messages. None of the approaches cover all criteria and this is the goal of the research reported in this paper (the last column in Table 1). Some of the existing CFMs are IRCFG (Inter-procedural Restricted Control-Flow Graph), CRE (Concurrent Regular Expressions), LGSPN (Labeled Generalized Stochastic Petri-Net), and Concurrent Control Flow Graph (CCFG). An IRCFG contains a set of Restricted CFGs (RCFGs), together with edges which connect these RCFGs. Each RCFG corresponds to a particular method and is similar to the CFG for that method, except that it is restricted to the flow of control that is relevant to message sending [12]. CRE was proposed as algebraic descriptions of the language of Petri nets by Garg et al. [20]. It is an extension of regular expressions with four operators: interleaving, interleaving-closure, synchronous composition and renaming. LGSPN is an extension to GSPN (Generalized Stochastic Petri-Net) where labels are assigned to Petri-net places and transitions. GSPN is itself an extension to Petri nets that allows both timed and immediate transitions [21]. Our CCFG model is further described in the rest of the article.

Table 1. Comparison of existing MBCFA techniques

	[11]	[12]	[13]	[14]	[15]	This work
1. Source of information	SD	RE-SD	SD	SD	SD	SD
2. UML version	UML 1.x	UML 1.x	UML 1.x	UML 1.x	UML 1.x	UML 2.0
3. Produced CFM	CRE	IRCFG	LGSPN	Petri-Net	None	CCFG
4. CFM's formalism	Set theory	By example	By example	By example	None	Metamodel
5. Transformation	By example	By example	Semi-formal[1]	Formal	None	Formal
6. Asynchronous message control flow	No (but can be extended)	No	Yes	Yes	Not clear	Yes
7. Inter-SD control flow	No	No	No	No	No	Yes
8. Loop	Yes	No	No	No	No	Yes
9. Condition	No	Yes	No	No	Yes	Yes
10. Polymorphism	No	No	No	No	No	Yes
11. CFP formalism	Yes	Yes	No	No	No	Yes

There has been another group of related works (see [22] for further details) which we can classify as CBCFA techniques with concurrency analysis. Most of these works are based on formal methods and are intended to be used by compilers mainly for program optimization purposes. Among the CFMs used, Task Interaction Graph (TIG) [23], i.e., a model that accounts for control flows within and between concurrent tasks, and TIG-based Petri-nets [24] are the closest models to our CCFG. More details on how they compare to our approach can be found in [22]. The work in [25] transforms Message Sequence Charts (MSC) (predecessors of UML SDs) to statecharts. This can not be easily applied to SDs since their new metamodel [10] has more complex constructs (such as loops and interaction occurrences) which makes their transformation to statecharts challenging. This, though, has to be further investigated.

[1] Plain English pseudo-code with some formal definitions.

1.2 Goals

The goals of our MBCFA approach are: (1) to devise a Control Flow Model (CFM), to handle concurrent control flow in SDs, (2) to present a set of formal mappings from an instance of SD to an instance of CFM, and (3) to propose a formalism to represent different concurrent CFPs.

Our technique assumes that the SDs and CDs (class diagrams) of a system's UML 2.0 [10] design model are available. The technique uses the given SDs and CDs to derive the control flow information of each SD. SDs will be used as the input for the derivation of control flow information, while CDs will be used to account for polymorphic messages in SD lifelines [22]. To perform a MBCFA technique based on UML 2.0, we propose a three-pronged solution: (1) A CFM referred to as Concurrent Control Flow Graph (CCFG); (2) A set of formal OCL-based mappings from an instance of the SD metamodel to an instance of the CCFG metamodel; (3) A formal representation of different CFPs of a CCFG, referred to as Concurrent CFPs (CCFPs), under the form of a grammar handling nested concurrent paths.

The rest of this paper is structured as follows. A Control Flow Model (referred to as CCFG), for MBCFA, is presented in Section 2. Section 3 presents a set of OCL-based mapping rules from SDs to CCFGs, and illustrates them on an example SD. Based on the CCFGs metamodel, Section 4 discusses how different control flow paths of a SD can be formally represented. Section 5 finally concludes this article and points out some of the future research directions.

2 Concurrent CFG: A Control Flow Model for SDs

One might argue that an intermediate CFM is not needed for MBCFA as an algorithm can be built to directly derive CFPs from a SD. However, there is a requirement that necessitates an intermediate CFM for SDs, as discussed below. Furthermore, in the context of MDA, and since the CFG is derived from UML sequence diagrams, the CFG model should be based on the UML notation and thus allow UML CASE tools to visualize/animate concurrent control flow in SDs. This can be useful in, for example, model execution, comprehension, and walk-through.

To better justify our choice, we first briefly identify the challenges of SDs' CFA in Section 2.1. Using the discussion in Section 1.1, in which we surveyed some of the existing CFMs in the literature, Section 2.2 explains our choice of a suitable CFM. We present the metamodel of our CFM in Section 2.3.

2.1 Challenges of SDs' CFA

Conventional CFA techniques [1] are usually applied to sequential programs, and are thus not easily applicable when concurrency has to be accounted for.

Asynchronous messages and the *par* interaction operator in SDs entail intra-SD concurrency. Fig. 1 shows an example SD that we will use later in the article to illustrate our approach. It contains two asynchronous messages, namely *addToQueue()* and *process()*, labeled C and E respectively. (The other messages are synchronous.) Fig. 1 also shows one of the new constructs in UML 2.0 SDs, namely combined fragments. The combined fragments are labeled *opt* and *loop*, which are respectively used to specify options (alternatives) and loops. The interaction occurrence (*ref*) is used to refer to other SDs. *par* is another combined fragment used to illustrate asynchronous communications of groups of messages: for instance, in Fig. 2, messages *m1*, and *m2* and *m3* are handled in parallel. The reader not familiar with those new constructs is referred to [10] for further details.

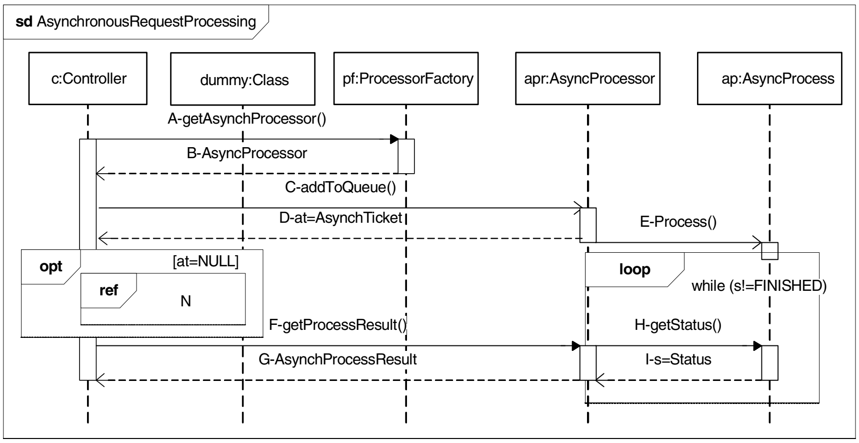


Fig. 1. A SD with asynchronous messages

Although traditional CFG models have constructs (i.e., nodes and edges) to specify branching and sequences of executions, they do not possess specific constructs to specify asynchronous messages or the concurrent sequences of executions.

2.2 Towards a CFM for SDs

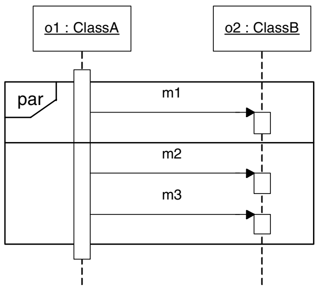


Fig. 2. A SD with *par* operator

Various CFMs have been proposed for CBCFA of concurrent programs in the literature, such as: [26], [27], [28], [29], [23] and [24]. The first three CFMs are used in the context of compilers, languages and formal methods. Although they are well-defined, they are difficult to adapt to the UML notation. Thus, they cannot be easily used in the context of UML and metamodel-based transformations. The work in both [23] and [24] reports on Petri net-based CFMs. Petri-nets [29] contain specific constructs to specify sequences of executions, either synchronous or not, where fork and join nodes (bars) are used to model synchronization of concurrent executions. One advantage of Petri-nets is that they have a well-established formal notation that has been widely used for the modeling of dynamic behavior, as well as extensive tool support.

Among the CFMs used in the previous MBCFA techniques (i.e., IRCFG [12], CRE [11], and LGSPN [13]), only LGSPN (a Petri-net based model) takes into account concurrent control flow.

UML has adopted a Petri-net like semantics for control and object flow modeling referred to as *Activity Diagrams* (AD). ADs have been in UML since its early 1.x versions . They account for both sequential and concurrent control flow and data flow. As UML 2.0 points out (Section 12.1 of [10]): "*These [UML activities] are commonly called control flow and object flow models.*" Among the alternative representations of LGSPN, TIG [23], TIG-based Petri-nets [24], Petri-net and UML AD, we choose the latter for the CFA of SDs. The reasoning for this decision is threefold:

1. AD already has a well-defined metamodel, which is needed by our MBCFA technique and satisfies our needs.
2. SD and AD are both in the context of UML. Both metamodels are part of a large collection (UML) and they have been designed in a similar methodology. Therefore, our technique may potentially benefit from the fact that both metamodels are part of the same context.
3. Furthermore, the generated CFM (a slightly modified version of ADs in our case) can be easily visualized/analyzed by standard UML 2.0 CASE tools.

UML 2.0 proposes six different but dependent activity packages (Section 12 of [10]): BasicActivities (*BA*), StructuredActivities (*SA*), IntermediateActivities (*IA*), CompleteStructuredActivities (*CSA*), ExtraStructuredActivities (*ESA*), and CompleteActivities (*CA*). Based on the analysis of the metamodel and class descriptions of the activity packages, we decide to use the *IA* package as the starting point towards a CFM for our MBCFA approach. The reasons for this selection are: (1) The *IA* package fits the needs of MBCFA, since it supports modeling both concurrency (by being Petri-net like, i.e., including ForkNode and JoinNode) and structured control constructs, and (2) The *IA* package is simpler than the other three packages (*CSA*, *ESA* and *CA*), as it does not include the modeling features needed for advanced data flow modeling. The *BA* and *SA* packages are not chosen since they are too simple, i.e. they do not include the ForkNode and JoinNode constructs. However, due to specific requirements in our MBCFA, which will be explained in the next section, we need to extend the *IA* package to a new activity package called *CCFG*.

2.3 Concurrent CFG Metamodel

We merge[1] the Concurrent CFG (CCFG) package from the *IA* package by adding new associations and sub-classes as a CFM for SDs. A CCFG (activity class in the metamodel) will be generated for one SD. In the case where a SD calls (refers to) another SD, there will be control flow edges connecting their corresponding CCFGs. We can refer to this concept as *Inter-SD CCFG*, similar to the concept of inter-procedural CFG [1].

The CCFG metamodel is shown in Fig. 3. Extensions are made to four of the classes in the *IA* metamodel: *Activity*, *ActivityNode*, *ExecutableNode*, and *Activity-Partition*, which are described next. Furthermore, since the *Activity* class of the *IA*

[1] Merge" is a terminology (stereotype) used on associations between two UML AD packages (Figure 175 of [10]). The classes of the AD package on the tail of the "merge" association extend the classes of the AD package on the head of the association.

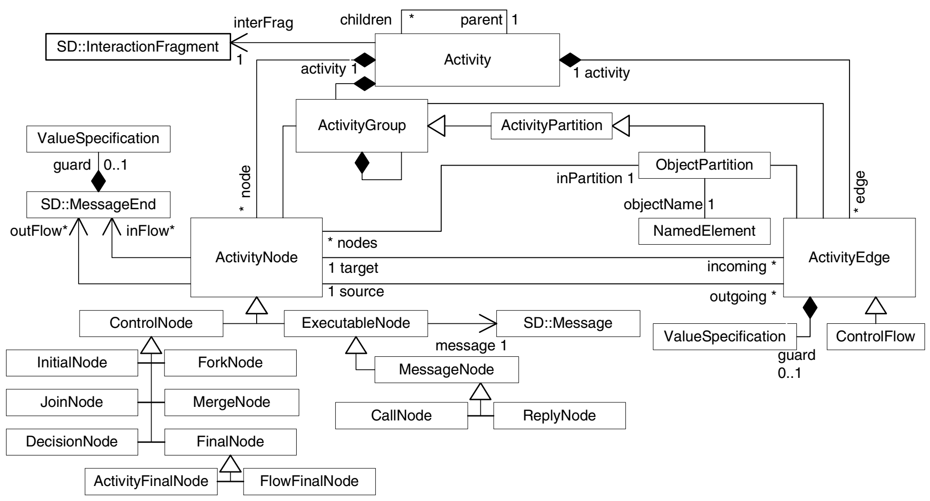


Fig. 3. CCFG metamodel

metamodel is extended, its sub-classes (*ControlNode*, *ExecutableNode* and their sub-classes as well) in the *CCFG* metamodel are also implicitly extended from their corresponding classes in the *IA* metamodel.

Each instance of the *Activity* class in the CCFG metamodel corresponds to an instance of the *InteractionFragment* class in the SD metamodel. Therefore, in order to access the interaction fragment associated with an activity, we need to add an association from the activity class in the CCFG metamodel to the interaction fragment class in SD metamodel. Note that the activity and CCFG classes of the CCFG metamodel are used interchangeably in this article. Furthermore, since SD interaction fragments can be nested, their corresponding activities have to be nested too. Therefore, we add a reflexive bidirectional association from the activity class to itself with role names *parent* and *child*. Each CCFG has one parent CCFG and can have multiple child CCFGs.

The need for an extension to *ActivityNode* arose when we started to design our mapping approach (Section 3). In order to build all control flows (edges) among all activity nodes of a CCFG, we need to make associations between activity nodes and their corresponding messages' message ends. In other words, we add two associations to *ActivityNode: inFlow* and *outFlow*, which are both targeted to the *SD::MessageEnd* class. These two associations will keep track of in and out flows of an activity node, to be used later in control flow connection. The reason why we assign a zero to many multiplicity (*) for these two associations is that there might be cases in which more than one in/out flows have to be built towards/from a node. Consider node *F* in Fig. 5 as an example, which has two in flows. Our strategy for control flow connection is to store send and receive events of each message in the in and out flow sets of its corresponding executable node. In case when a message is inside an *alt* or *loop* combined fragment, the message ends are stored inside in and out flows of the decision node, responsible for controlling the flow of the combined fragment. The guard association of the *MessageEnd* is intended for storing guard conditions when storing in/out flows of a node. The guard will be later copied to the corresponding activity edge. More details on how the in and out flows are handled are provided in Section 3 and [22]. Note that the subclasses of *ActivityNode* in the CCFG metamodel extend *ActivityNode* in similar ways like subclasses of *ActivityNode* in the IA metamodel. For example, *CCFG::ControlNode* extends *CCFG::ActivityNode* (which itself extends *IA::ActivityNode*) and *IA::ControlNode*. We also change the semantic of activity final nodes in a way that they can have outgoing edges. This is needed to be done since when a SD calls another SD using an interaction occurrence, there needs to be a control flow from the activity final node of the CCFG corresponding to the called SD to a message node in the CCFG corresponding to the caller SD (see Fig. 5 for example).

We define new *CallNode* and *ReplyNode* classes in a CCFG which correspond to a call/reply message in the corresponding SD. Distinguishing between call and reply messages (and their corresponding activity nodes) can enrich our MBCFA technique (Section 4). These two classes in a C CFG are generalized by a new abstract class *MessageNode* which itself is being generalized by *ExecutableNode* in *IA*. One more extension is to make it possible to access the corresponding message of an *ExecutableNode*. In order to do this, we add an association to *ExecutableNode*, which is entitled *message* and is targeting the *SD::Message* class.

An extension to *ActivityPartition* is made by adding a subclass named *ObjectPartition*. This is to group a CCFG's activity nodes based on the receiver object of their corresponding messages: swimlanes in an AD then correspond to classifiers in the corresponding SD.

3 Consistency Mapping Rules from SDs to CCFGs

Using OCL [30], we propose a consistency rule-based approach to map SDs into CCFGs. The mapping rules are useful in several ways: (1) they provide a logical specification and guidance for our transformation algorithms that derive a CCFG from a SD (both being instances of their respective metamodels), and (2) they help us ensure that our CCFG metamodel is correct and complete with respect to our control flow analysis purpose, as the OCL expression composing the rules must be based on the metamodels.

3.1 SD Metamodel

Since our technique uses the SD metamodel as the source, we extract the SD metamodel from the UML specification [10].

After omitting unnecessary details, and showing only those feature that are of interest to us (e.g., role names, multiplicities), we obtain the class diagram in. In the SD metamodel, *Interaction* is the class from which SD instances are instantiated. Each *Interaction* has a set of *Message*'s whose orders are indicated by instances of *GeneralOrdering,* via *MessageEnd*'s. *CombinedFragment* is a construct which is used to-model loops, conditions, parallel sequence of messages, etc. A *CombinedFragment* can have one or more *InteractionOperand*. The rest of the class descriptions can be found in Section 14 of [10].

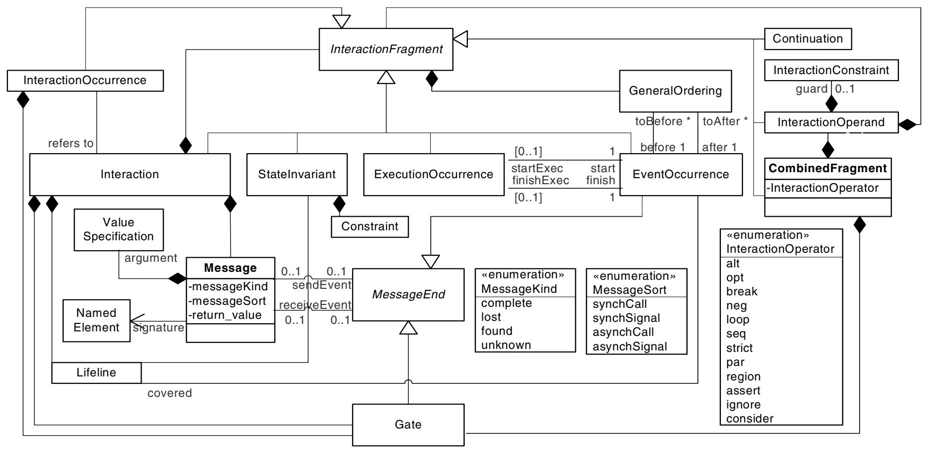


Fig. 4. UML 2.0 sequence diagram metamodel

3.2 Consistency Rules

We have derived fourteen consistency rules, expressed in OCL, that relate different elements of an instance of a SD metamodel to different elements of an instance of the CCFG metamodel. They are all listed in Table 2 and are illustrated them with the example SD of Fig. 1. However, due to space constraint, we only describe a subset of them in the remaining sections.

To demonstrate the feasibility of our approach, we have applied the consistency rules to the SD of Fig. 1 and the resulting CCFG is shown in Fig. 5. Each message node in CCFG of Fig. 5 is labeled with the corresponding message name in SD of Fig. 1. Two fork nodes in the CCFG are created because of the two asynchronous messages in the SD of Fig. 1. Due to space limitations, we describe in the next sections only two of the rules (#2 and #3) and how they are applied to the SD of Fig. 1. In the rule descriptions below, symbol → means mapping from a SD feature to a CCFG feature. Further details on applying each consistency rule and the OCL expressions of the other consistency rules are described in [22].

Table 2. Mapping rules from SDs to CCFGs

#	SD feature	CCFG feature
1	Interaction	Activity
2	First message end	Flow between InitialNode and first control node
3	SynchCall/SynchSignal	CallNode
4	AsynchCall or AsynchSignal	(CallNode+ForkNode) or ReplyNode
5	Message SendEvent and ReceiveEvent	ControlFlow
6	Lifeline	ObjectPartition
7	*par* CombinedFragment	ForkNode
8	*loop* CombinedFragment	DecisionNode
9	*alt/opt* CombinedFragment	DecisionNode
10	*break* CombinedFragment	ActivityEdge
11	Last message ends	Flow between end control nodes and FinalNode
12	InteractionOccurrence	Control Flow across CCFGs
13	Polymorphic message	DecisionNode
14	Nested InteractionFragments	Nested CCFGs

3.2.1 First Message End → Flow Between InitialNode and First Control Node

This consistency rule checks if there is a flow from the initial node of every CCFG to its first control node. The first control node of a CCFG is the one that corresponds to the first message of the corresponding SD.

OCL Mapping

```
1  SD::InteractionFragment.allInstances->forAll(interFrag:InteractionFragment|
2     CCFG::InitialNode.allInstances->exits(in:InitialNode|
3        in.activity=Utility::Util.getCCFG(interFrag) and
4        in.outgoing->includes(flow:ControlFlow|
5           getCCFG(interFrag).node->exits(an:ActivityNode|
6              an.inFlow->includes(Utility::Util.getFirstMessage
7              (interFrag).sendEvent) and flow.target=an
8           )
9        )
10    )
11 )
```

The CCFG of each interaction fragment is checked to have an initial node (lines 1-2) with specific characteristics. There should be a control flow from the initial node to the activity node having the *sendEvent* of the first message of the interaction fragment in its *inflow* (lines 4–7).

To reduce the complexity of our consistency rules, we have defined several utility functions inside a utility class *Util*, such as *getCCFG()* and *getFirstMessage()*, as they are used in the above rule. *getCCFG()* returns the *CCFG::Activity* instance associated with an instance of *SD::InteractionFragment*. *getFirstMessage()* returns the first message of a given interaction fragment according to the ordering provided by its *GeneralOrdering*. *GeneralOrdering* is the SDs mechanism to order messages. More details are provided in [22].

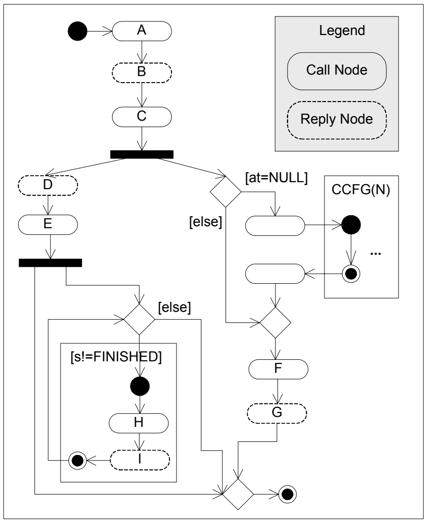


Fig. 5. CCFG of the SD in Fig. 1

Example

Fig. 6-(a) shows part of the CCFG instance that satisfies the above consistency rule based on Fig. 1. The corresponding part of the resulting CCFG is represented in Fig. 6-(b). Executable node *enA* corresponds to the message *A* (*getAsynchProcessor)* in Fig. 1. *enAse* is the *sendEvent MessageEnd* of the message *A*. *enAse* is already in the inflow of node *enA* because of the rule #3. Furthermore, *getFirstMessage(ccfg)* returns message *A,* and in this way, the appropriate control flow connection between the *ccfg*'s initial node and node *enA* is checked to exist.

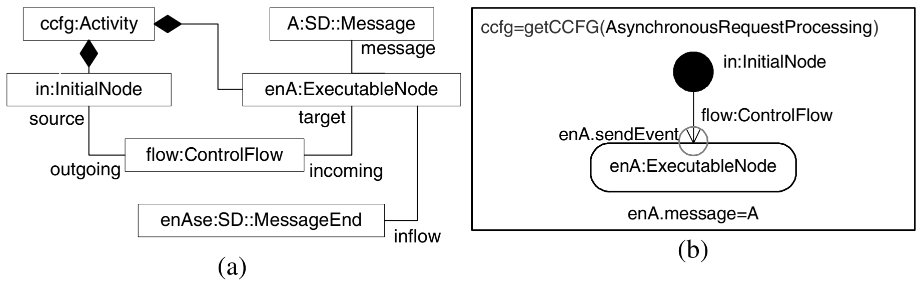


Fig. 6. (a)-Part of the CCFG instance *ccfg* mapped from the SD in Fig. 1, satisfying the consistency rule #2. (b)-Part of the CCFG, corresponding to the instance shown in (a).

3.2.2 SynchCall/SynchSignal→CallNode

This rule maps synchronous (call or signal) messages of a SD to call nodes of a CCFG. (These messages are identified thanks to enumeration values *synchCall* and *synchSignal* of message attribute *messageSort*.)

OCL Mapping

```
1  SD::Message.allInstances->forAll(m:Message|
2     (m.messageSort=SD::MessageSort.synchCall) or
3     (m.messageSort=SD::MessageSort.synchSignal)
4     implies
5        CCFG::CallNode.allInstances->exits(cn:CallNode|
6           cn.message=m and
            -- check object partition
7           cn.inPartition= Utility::Util.getObjectPartition(
               m.receiveEvent.covered.connectable_element_name) and
            -- make sure cn is prepared for control flow (edge) connections
            -- m.SendEvent/m.ReceiveEvent should be in inFlow/outFlow of cn
8           cn.inFlow->includes(m.sendEvent) and
9           cn.outFlow->includes(m.receiveEvent) and
10          Utility::Util.getCCFG(m.interaction).node->includes(cn)
11       )
12 )
```

The *synchCall* and *synchSignal* messages of a SD are selected in lines 1-3. The existence of a corresponding call node *c* is checked in lines 5-10. The call node *cn*'s message association value should be *m* (line 6), its object partition should be lifeline of message *m* (line 7), and its inflow and outflow sets should include only *m*'s send and receive events, respectively (lines 8-9). These inflow and outflow information are needed since the control flow consistency rule (#5) will use them to connect nodes to each other to form the control flow. Line 10 makes sure that node *cn* is a part of the CCFG corresponding to interaction containing message *m*. *getObjectPartition()* is another utility function, which returns the object partition instance associated with a lifeline (using consistency rule #6).

Example

Fig. 7-(a) shows part of the CCFG instance corresponding to message *A* in Fig. 1 that satisfies the consistency rule #3. The corresponding part of the resulting CCFG is represented in Fig. 7-(b). Call node *cn* corresponds to message *A* in Fig. 1. Since the receiver lifeline of message *A* is *pf:ProcessorFactory*, the *inPartition* association of *cn* corresponds to *ObjectPartition* instance *op* which is associated with object and class names *pf* and *ProcessorFactory*, respectively.

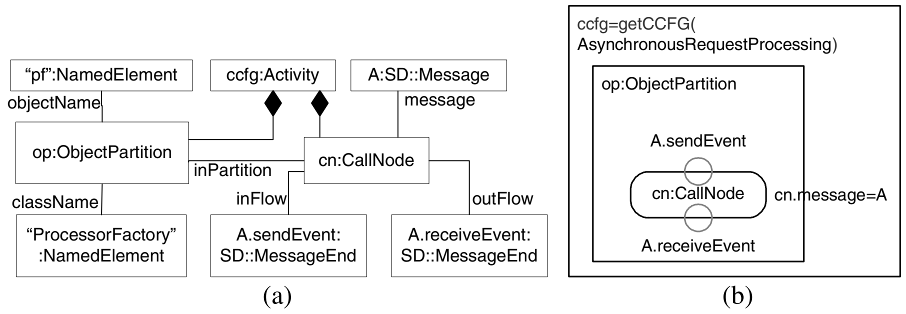


Fig. 7. (a)-Part of the CCFG instance *ccfg* mapped from the SD in Fig. 1, satisfying the consistency rule #3. (b)-Part of the CCFG, corresponding to the instance shown in (a).

4 Concurrent Control Flow Paths

The CCFG corresponding to a SD includes one or more Concurrent Control Flow Paths (CCFPs). Each CCFP is defined as a concurrent path, starting from the initial node of the CCFG to its activity final node. Concurrent paths include *all* the branches going out from a fork node in a CCFG. A CCFP is derived by traversing from the initial node to the final node and concatenating all the nodes in the path, in order. Similar to conventional CFA techniques, special considerations regarding decision nodes should be made in terms of conditions and loops, i.e., two different CCFPs for true/false edges of a conditional should be derived. Formally, the set of all CCFPs of a CCFG can be represented using the grammar [31] in Fig. 8.

$$
\begin{aligned}
CCFP &::= CCFP.CCFP \mid FlowNode \mid \begin{pmatrix} CCFP \\ \cdots \\ CCFP \end{pmatrix} \mid LoopCCFP \mid ConditionalCCFP \mid \varepsilon \\
FlowNode &::= InitialNode \mid FinalNode \mid CallNode \mid Re\,plyNode \mid ForkNode \mid JoinNode \\
LoopCCFP &::= CCFP \mid (CCFP)^{avg} \mid (CCFP)^{max} \mid \varepsilon \\
ConditionalCCFP &::= TrueCCFP \mid FalseCCFP \\
TrueCCFP &::= CCFP \\
FalseCCFP &::= CCFP
\end{aligned}
$$

Fig. 8. Grammar defining the set of CCFPs of a CCFG

In the grammar shown in Fig. 8, the first line indicates that: (1) the concatenation of two CCFPs gives a CCFP, (2) *FlowNode* instances are the basic building blocks for CCFPs, and (3) the notation with two parentheses is for CCFPs with intra-SD concurrency. When there are fork and join nodes in a CCFG, it means that there is intra-SD concurrency in the CCFG: different paths between the fork and join nodes can be executed concurrently. An open (close) parenthesis corresponds to a fork (join) node. *LoopCCFP* and *ConditionalCCFP* denote that a CCFP can be built from loop and conditional blocks of a CCFG. ε is the standard empty string notation, meaning an empty CCFP in our context.

The third line in Fig. 8 is for loop CCFPs and has a strategy similar to what is used to test loops in code, i.e. each loop is bypassed (ε) – if possible, taken only once (CCFP), a representative or average number above one (*avg*), and a maximum number of times (*max*). The fourth line is for conditional CCFPs and specifies that both the true or false path of a condition must be followed to derive two different CCFPs. The fifth and sixth lines state that the true and false paths of a conditional CCFP are CCFPs as well.

For example, our representation of CCFPs of a CCFG can be useful for test requirement generation applications and data flow analysis purposes (such as *def-use* pairs).

A constraint for a well-formed CCFP is that the first and the last flow nodes of a CCFP should be the initial and activity final nodes of the CCFG corresponding to an interaction (SD). As examples of CCFPs, considering the CCFG in Fig. 5, the set of all CCFPs is shown in Fig. 5, assuming the average and maximum number of times one wants to exercise the loop are 2 and 3, respectively. The ρ_i symbols are used to refer to distinct CCFPs.

$$\rho_1 = ABC\left(\begin{matrix} DE\left(\begin{matrix} \\ \end{matrix}\right) \\ FG \end{matrix}\right) \qquad \rho_2 = ABC\left(\begin{matrix} DE\left(\begin{matrix} HI \\ \end{matrix}\right) \\ FG \end{matrix}\right)$$

$$\rho_3 = ABC\left(\begin{matrix} DE\left(\begin{matrix} (HI)^2 \\ \end{matrix}\right) \\ FG \end{matrix}\right) \qquad \rho_4 = ABC\left(\begin{matrix} DE\left(\begin{matrix} (HI)^3 \\ \end{matrix}\right) \\ FG \end{matrix}\right)$$

Fig. 9. CCFPs of the CCFG in Fig. 5

5 Conclusions

This article presents a control flow analysis methodology for UML 2.0 sequence diagrams, which is based on defining formal mapping rules between metamodels. The output of our technique can be used in Sequence Diagram-based test techniques. Other usages of the generated control flow information include model execution, model comprehension, and conformance verification of model with code. Based on well-defined UML 2.0 activity diagrams, we used a customized activity diagram metamodel, referred to as Concurrent Control Flow Graph (CCFG), as the control flow model in this work. Furthermore, a grammar was defined for Concurrent Control Flow Paths (CCFPs). CCFPs are a generalization to the conventional Control Flow Path concept handling nested concurrent paths. Our entire approach was applied to an example SD and is further described and illustrated in [22].

Some of our future research directions are: (1) definition of test-based coverage criteria for CCFGs and CCFPs, (2) a metamodel for CCFPs and algorithms to derive CCFPs from a CCFG, (3) more investigation on the proposed grammar for CCFPs and ways to automate the CCFPs derivation process by building a parser, (4) defining a consistency-rule based Data Flow Analysis (DFA) technique to derive data flow information from SDs, (5) tool support, and (6) implementing OCL transformation rules to transform a SD into a CCFG, i.e., using MDA's [19] transformation definition language. The last idea is similar to the transformation rules given in [19] to transform a PIM (Platform Independent Model) to a PSM (Platform Specific Model) in the context of the MDA framework.

References

[1] S. Muchnick, *Advanced Compiler Design and Implementation*, First ed: Morgan Kaufmann, 1997.

[2] A. T. Chusho, "Test data selection and quality estimation based on the concept of essential branches for path testing," *IEEE Tran. on Soft. Eng.*, vol. 13, no. 5, pp. 509-517, 1987.

[3] A. Bertolino and M. Marre, "Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs," *IEEE Tran. on Soft. Eng.*, vol. 20, no. 12, pp. 885-899, 1994.

[4] S. Rapps and E. J. Weyuker, "Data flow analysis techniques for test data selection," Proc. Int. Conf. on Soft. Eng., pp. 272-278, 1982.

[5] M. Harrold and M. Soffa, "Interprocedual data flow testing," Proc. Symp. on Soft. Testing, Analysis, and Verification, pp. 158-167, 1989.

[6] B. Marick, "Experience With the Cost of Different Coverage Goals For Testing," Proc. Pacific Northwest Soft. Quality Conf., pp. 147-164, 1991.

[7] Z. Jin and J. Offutt, "Coupling-based Criteria for Integration Testing," *Soft. Testing, Verification, and Reliability*, vol. 8, no. 3, pp. 133-154, Sept. 1998.

[8] OMG, "Unified Modeling Language Specification (v1.3)," 1999.

[9] OMG, "Unified Modeling Language Specification (v1.5)," 2003.

[10] OMG, "UML 2.0 Superstructure Final Adopted specification," 2003.

[11] M. Okazaki, T. Aoki, and T. Katayama, "Formalizing sequence diagrams and state machines using Concurrent Regular Expression," Proc. Int. Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, pp. 74-79, 2003.

[12] A. Rountev, S. Kagan, and J. Sawin, "Coverage Criteria for Testing of Object Interactions in Sequence Diagrams," Proc. Conf. Fundamental Approaches to Soft. Eng., pp. 289-304, 2005.

[13] S. Bernardi, S. Donatelli, and J. Merseguer, "From UML sequence diagrams and statecharts to analyzable Petri-net Models," Proc. Int. Workshop on Soft. and Performance, pp. 35-45, 2002.

[14] J. Cardoso and C. Sibertin-Blanc, "Ordering actions in sequence diagrams of UML," Proc. Int. Conf. on Information Technology Interfaces, pp. 3-14, 2001.

[15] E. Burd, D. Overy, and A. Wheetman, "Evaluating Using Animation to Improve Understanding of Sequence Diagrams," Proc. Int. Workshop on Program Comprehension, pp. 07-113, 2002.

[16] Y. Wu, M.-H. Chen, and J. Offutt, "UML-Based Integration Testing for Component-Based Software," Proc. Int. Conf. on COTS-Based Software Systems, pp. 251-260, 2003.

[17] A. Abdurazik and J. Offutt, "Using UML Collaboration Diagrams for Static Checking and Test Generation," Proc. Int. Conf. on the Unified Modeling Language, pp. 383-395, 2000.

[18] F. Fraikin and T. Leonhardt, "SeDiTeC-testing based on sequence diagrams," Proc. Int. Conf. on Automated Soft. Eng., pp. 261-266, 2002.

[19] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture-Practice and Promise*, 1st ed: Addison-Wesley Prof., 2003.

[20] V. K. Garg and M. T. Ragunath, "Concurrent regular expressions and their relationship to Petri nets," *Theoretical Computer Science*, vol. 96, no. 2, pp. 285-304, 1992.

[21] S. Donatelli and G. Franceschinis, "PSR Methodology: integrating hardware and software models," Proc. Int. Conf. in Application and Theory of Petri Nets, pp. 133-152, 1996.

[22] V. Garousi, L. Briand, and Y. Labiche, "Control Flow Analysis of UML 2.0 Sequence Diagrams," Technical Report SCE-05-09, Carleton University, http://www.sce.carleton.ca/squall/pubs/tech_report/TR_SCE-05-09.pdf, 2005.

[23] D. L. Long and L. A. Clarke, "Task interaction graphs for concurrency analysis," Proc. Int. Conf. on Soft. Eng., pp. 44-52, 1989.

[24] A. T. Chamillard and L. A. Clarke, "Improving the accuracy of Petri net-based analysis of concurrent programs," Proc. Int. Symp. on Soft. testing and analysis, pp. 24-38, 1996.

[25] I. Krüger, R. Grosu, P. Scholz, and M. Broy, "From MSCs to Statecharts," Proc. Int. Workshop on Distributed and parallel Embedded Systems, pp. 61-71, 1999.

[26] H. R. Nielson and F. Nielson, "Infinitary Control Flow Analysis: a Collecting Semantics for Closure Analysis," Symp. on Principles of Programming Languages, pp. 332-345, 1997.

[27] J. Bauer, "A control-flow-analysis for multi-threaded java with security applications," Master's thesis, Universitat des Saarlandes, 2001, pp. 97.

[28] P. D. Blasio, K. Fisher, and C. Talcott, "A Control-Flow Analysis for a Calculus of Concurrent Objects," *IEEE Trans. on Soft. Eng.*, vol. 26, no. 7, 2000.

[29] W. Brauer, W. Reisig, and G. R. (eds.), "Petri nets, central models and their properties," *LNCS*, vol. 254, 1987.

[30] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*: Addison Wesley, 2003.

[31] G. Rozenberg and A. Salomaa, *Handbook of Formal Languages*, 1st ed: Springer, 1997.

Designing a Domain-Specific Contract Language: A Metamodelling Approach

Zhan En Chan[1] and Richard F. Paige[2]

[1] Department of Computer Science, University of Warwick, UK
echan@dcs.warwick.ac.uk
[2] Department of Computer Science, University of York, UK
paige@cs.york.ac.uk

Abstract. Domain-specific languages are of increasing importance in software engineering. Little attention has been paid to the systematic production of domain-specific contract languages (DSCLs). In this paper, we present a metamodel-based approach for designing DSCLs. An extensible metamodel for software contracts is presented, and a process for building DSCLs is sketched. Finally, an example of building a DSCL is demonstrated, using the metamodel and process.

1 Introduction

Software contracting has been proposed as a useful technique to improve software reliability, particularly, but not exclusively, in object-oriented (OO) software development, usually in the form of pre- and postconditions on methods. The strengths and weaknesses of contracting are widely acknowledged in the literature, e.g., [20,22]. The key mechanism in software contracting is the *assertion*. Assertions have been used for many years in software development [34], and rich mathematical theories have been developed [10].

Since their introduction, software contracts have been used in many different ways – and to different purposes – in different software projects. A wealth of contract languages have been developed, and tool support for some of these languages is now mature. Unfortunately, the sheer number of different contract languages and approaches for using them create a barrier for component integration and reuse, for developing software contract tool support, and for improving existing languages. Moreover, with the growing move towards domain-specific languages [11], it is becoming increasingly difficult to see how to systematically inject contracts into these languages. This could be simplified by the existence of domain-specific contract languages (DSCLs), and techniques for helping to construct DSCLs, particularly a *development process* and a *metamodel* explaining existing types of contracts and their relationships.

A general, broad, though incomplete classification of software contracts was presented by Beugnard [5]. It is incomplete for several reasons: it omits certain types of contracts (e.g., those for aspect-oriented modelling), and some types of domain-specific contracts (e.g., unit measurement in safety critical systems) cross several categories (e.g., unit contracts are neither purely syntactical nor behavioural).

The lack of a systematic classification of software contracts makes it difficult to develop software contract tool support, determine which kinds of contracts are useful for particular development projects, and impedes domain-specific contract language design. To help to remedy this situation, we present an extensible metamodel for software contracts, along with a process for eliciting contracts, which can be used for producing DSCLs. The benefits of metamodelling are informally acknowledged in [1]. In our case, a metamodelling approach has the following advantages:

1. It helps software engineers identify and apply software contracts more efficiently, since the metamodels help to structure and classify the different types of contracts and the ways in which they can be applied.
2. The metamodels introduced are extensible. This gives an account for producing new types or uses of software contracts beyond those currently identified. A process for building and extending the metamodels is introduced in Section 3.1.

The remainder of this paper is organised as follows: Section 2 describes the different types of software contracts used in practice. Section 3 discusses the metamodelling approach, and particularly details four views in the metamodel for software contract. Section 4 demonstrates how the metamodelling approach can be used to help in the process of building a domain-specific contract language.

2 Background

In this section, the current uses of software contracting in systems engineering will be discussed, in terms of *contract binding* (i.e., when and how contracts are bound to software entities), *contract representation*, and *domain of applicability*. This will be used to generate different views in the contract metamodel, and can also be used to precisely define a contract:

Definition 1. ***[Contract].*** *A contract is a set of one or more assertions that are expressed in a language (ideally with a precise specification of its syntax and semantics), are bound to one or more software entities, and are used in a specific domain of applicability.*

2.1 Contract Binding

The key element in contracts, the assertion, has been used in structured methodologies, e.g., [34,30]. Contracts have also been successfully used to specify aspect composition in Aspect-Oriented Programming (AOP) [13]. More recently, software contracts have been used in event-driven software systems for event specifications [17,21], architectural models [28,26], and components. We argue that a software contract can be *bound* or *attached* to any software artifact (e.g., procedures, classes, aspects, events) in any modelling language. The meaning, importance, and value of a binding will of course be dependent on the modelling technique and how it is used. This binding or attachment can be done *statically* (i.e., at software construction time) or *dynamically* (when the software executes).

2.2 Contract Representation

A representation is a logical category of contracts which have similar syntax, degree of formality, and degree of expressiveness. There are four different types of representations for software contracts [7], though distinctions are not meant to be exact:

1. mathematical languages (e.g., Z, OCL); these can be further categorised into languages that support proof (e.g., Z) and those that do not (e.g., OCL).
2. semi-formal modelling languages (e.g., UML)
3. programming languages (e.g., C#, Java)
4. natural languages (e.g., comments in program source code, requirements)

There are limitations in using particular contract representations in different situations. This depends on the purpose behind using the contract, the level of formality, and when the contract is used in the software development lifecycle (e.g., to drive testing).

2.3 Domain of Application

Domain-Specific Software Engineering has received much attention in recent years; it advocates tailored approaches to address domain-specific concerns. Domain-specific solutions are common in application domains such as safety-critical systems, Web Services, and mobile computing. Numerous examples of contract languages constructed to work in specific domains can be found in the literature, e.g., [8,3,9,32,15].

Fig. 1 shows an example of a domain-specific contract for a poll function in the Cooling Control of an IMA system [27]. The precondition of the contract requires an input in units of kg^2 and with an accuracy of 10^{-6}, whilst the postcondition promises to return a result in unit of kg if the precondition is satisfied. The *accuracy* and *units* contracts enrich OCL modelling in the safety-critical domain with data accuracy and measurement units. A more detailed survey on contracts in different application domains is available in [7].

```
context Cooling Control::poll(x:float):float
pre: accuracy(x)>=10^(-6) and units(x)==kg2
post: result*result == x and units(result)==kg
```

Fig. 1. A contract in a safety-critical application (extracted from [27])

3 A Metamodel for Contracts

We now turn to the two key contributions of this paper: an extensible metamodel for software contracting approaches, and a description of the engineering process used for constructing the metamodel. This process can in turn be used to elicit new types of contracts, which can then be integrated into the metamodel.

The metamodel for contracts is derived from the results of an extensive survey [7] on contracts and their applications. It is described in UML annotated with OCL; as such it should be accessible and understandable to a substantial proportion of the Model-Driven Development community.

We first present the engineering process that we used to elicit contracts and construct the contract metamodel. This process will be useful for engineers who wish to extend the contract metamodel that we provide, and produce their own DSCL instantiated from the metamodel.

3.1 Contract Elicitation and Analysis Process

The metamodel for contracts is the result of a *Contract Elicitation and Analysis Process (CEAP)*. As its name suggested, the aim of the CEAP is to elicit and analyse contracts in order to understand their key attributes: their binding, their representation, and their domain of applicability. The CEAP is an iterative process (Figure 2). There are three main activities: Elicitation, Analysis, and Classification. Each cycle in the CEAP enriches the existing taxonomy in terms of one or more of the key attributes of interest. For instance, a cycle may focus on safety critical applications; another may focus on contracts that are used in different modelling techniques.

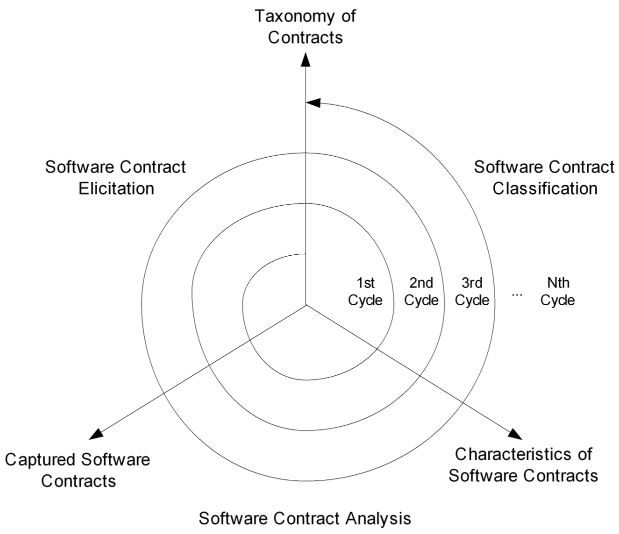

Taxonomy of Contracts
Software Contract Elicitation
Software Contract Classification
1st Cycle
2nd Cycle
3rd Cycle
...
Nth Cycle
Captured Software Contracts
Characteristics of Software Contracts
Software Contract Analysis

Fig. 2. Contract Elicitation and Analysis Process

CEAP is derived from the spiral software development model [33] and the spiral model in requirements engineering [14]. The main difference is that CEAP produces metamodels and taxonomies, rather than software or requirements.

The advantage of defining and using CEAP is that it gives extensibility to its product: the metamodel for contracts. This is very useful for DSCL design, as new types of contracts can emerge from new application domains for new purposes. In other words, the metamodel can be kept up-to-date by performing additional CEAP cycles.

3.2 Overview of the Metamodel

We now present our contract metamodel, derived from applications of CEAP cycles, starting with an architectural package diagram. As shown in Fig. 3, there are six packages in the metamodel:

1. *SoftwareContract* represents the concept of a software contract; this will be specialized in different views and representations in the metamodel. Essentially this package represents common behaviours of software contracts, e.g., that their assertions evaluate to $true$ or $false$ based on some state binding. Additional details are considered below.
2. *Representation* encapsulates those classes that are used to represent a software contract's abstract or concrete syntax (see Section 2.2).
3. *ModelTechView, EngTaskView, DomainView, NegLevelView* represent four views of software contract, which are extensions of *SoftwareContract*. These are explained in more detail in successive sections. Some of these views encompass the categories of [5], but there are new views capturing domains and engineering tasks as well.

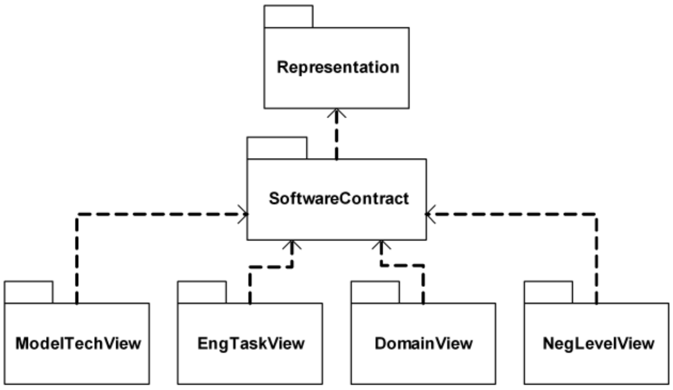


Fig. 3. Software Contract Metamodel

Dependencies among these packages are straightforward; the dependencies are implemented by generalisations: each package contains a top-level class that generalises $Contract$, thus producing a simple, flat taxonomy of contracts. Additional details will be added in the following sections.

3.3 An Abstract Software Contract

We now present the details of the Software Contract package, which is used to capture the essential characteristics of contracts. This metamodel will in turn be specialised and generalised to capture different representations and views in subsequent sections. A generic, yet extensible architecture is more accessible to study, analyse, and use for building DSCL and produce tool support for existing contract languages. Based on this claim, we constructed a metamodel derived from the framework found in Design-by-Contract [19], which is based on assertions. Most contracts in the literature are based on such a foundation (e.g., OCL), which supports conceptual elements found in assertions (i.e., preconditions, postconditions, invariants).

As shown in Fig. 4, there are three different mechanisms which define software contracts:

1. *ExceptionHandlingPolicy* is a set of mechanisms which are used to handle contract violations [5,18,20]. There are five different types of handling policy: reject, retry, ignore, wait, and negotiate. Note that this model supports selecting and sequencing of policies; in some cases it may be necessary to identify a unique exception handling policy.
2. *StaticAnalysis* checks that a contract is satisfied at compile time; the default is that the contract is satisfied at run-time, so this mechanism may be empty.
3. *Assertion* evaluates to $true$ or $false$ at runtime. A $false$ evaluation will result in invoking the exception handling policy noted previously.

Detailed discussion of contract checking and enforcement mechanisms is out of the scope of this paper; we refer the reader to the literature [5,20,7].

As well as capturing enforcement mechanisms, Fig. 4 also depicts the relationships among the elements of a typical software contract. A *Contract* may consist of many *Constraint*s. These constraints may bind to various artifacts depending on the modelling technique that is in use. A *Contract* may consist of sub-contracts [20] or depend on another contract [23,24]. There are seven types of constraints that can be associated with a contract, namely *Pre* (precondition), *Post* (postcondition), *ClassInv* (class invariant)[1], *LoopInv* (loop invariant), *LoopVar* (loop variant), *Guard* (conditional logic), and *Interface* (component interface). However, only the first six constraints are used as assertions and are verifiable at runtime.

In Beugnard's paper [5], an interface is the simplest form of contract. Though an *Interface* is not used in assertions, we argue that an interface contract is a constraint of a component rather than a type of contract.

The metamodel for an abstract software contract is not yet complete. For instance, it is currently possible to describe a contract that has two interfaces; or a contract that contains itself. Hence, we need to add OCL constraints to the metamodel. The necessary expressions are shown in Fig. 5.

3.4 Four Additional Views

The typical contract metamodel in Section 3.3 only depicts the common elements in contracts; it is insufficent for reasoning about contracts for special purposes, e.g., domain-specific contracts which may add further information pertaining to a domain of application. In this section, we extend the typical contract metamodel from four different perspectives. Although these views are apparently disjoint, they are inter-related and can be used simultaneously to construct a language and reason about a contract. A detailed example of applying all four views to reason about a software contract can be found in (p.56-60, [7]).

[1] UML 2.0 also supports StateInvariant which is similar in intent to ClassInv.

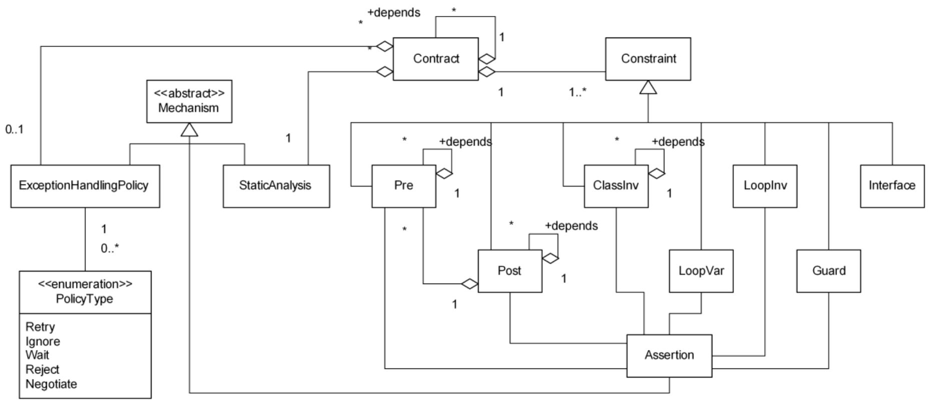


Fig. 4. A Metamodel for an Abstract Software Contract

```
context Contract inv:
self.depends.forall(c : Contract | c <> self)
self.constraint.select(oclIsTypeOf(Interface)).size() == 1
```

Fig. 5. Additional OCL constraints

View of Application Domains. Domain-specific contracts usually contain charaterised counterparts of *area of concerns* from their application domain. For example, measurement units in safety-critical systems [31,27], information flow in web applications [3], location and itinerary constraints in mobile agent systems [15] and end-user software engineering [4]. Some of the areas of concern overlap with others (e.g., security [8] and safety). This is due the nature of shared information in the application domain (e.g., information used for certifying safety, such as hazards and risks, may also be of importance in certifying security). As a result, the domain-specific contract is an extension of a typical contract containing domain-specific contraints (Fig. 6). This architecture has numerous advantages:

1. It is simple to extend in order to accommodate domain-specific contraints that are extracted from contracts in unexplored domains.
2. It can accommodate interdisciplinary or intra-domain contracts.

Due to space limitations, the OCL constraints for this view are omitted here; they can be found in (p.51-52, [7]).

View of Engineering Tasks. In the view of engineering tasks, we emphasize the identity of contracts pertaining to different software engineering tasks. Evidence [22,20,12] shows that contracts contribute to various engineering tasks, from requirements documentation to system deployment to certification. They also tend to appear in various representations and structures in different engineering tasks [16,6]. Typically, there are five major tasks in software development life cycle: requirements gathering, design, implementation, testing, and deployment [33]. There may be additional tasks in some

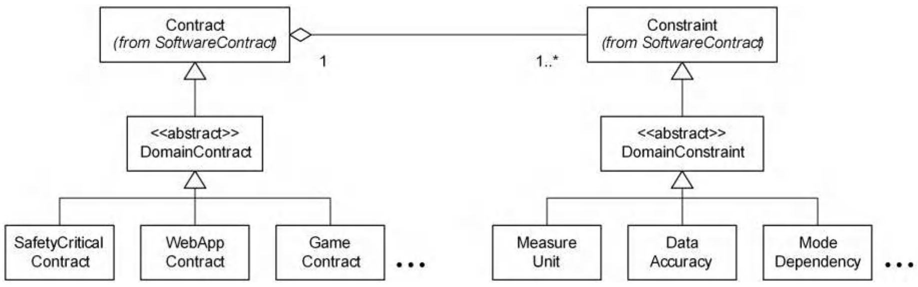


Fig. 6. View of Application Domains

situations, e.g. usability evaluation. As yet, there is no evidence that software contracting is used in these tasks.

As shown in Fig. 7, a contract can be used in various engineering tasks, in different representations. On the other hand, an engineering task may use a number of different contracts for specifying a software component. Five major engineering tasks are specialised under *EngTask*. Fig. 7 cannot articulate which representation is or is not available to a given engineering task; some of these decisions are domain-specific (e.g., whether a programming language can be used to provide contracts acceptable for deployment in a safety critical domain), but others can be made at this time. Therefore, we must supply OCL constraints; these are omitted and found in [7].

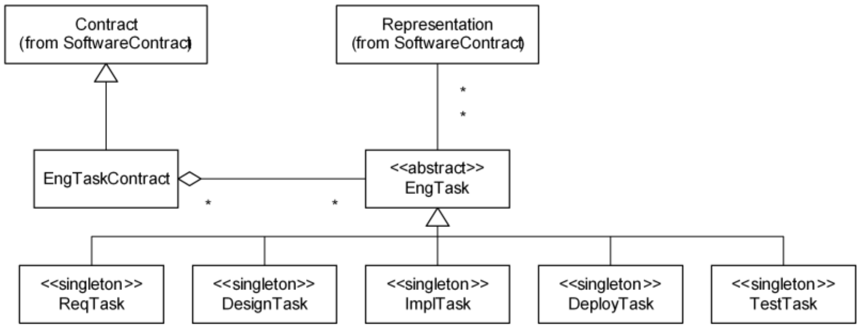


Fig. 7. View of Engineering Tasks

View of Modelling Techniques. Contracts are bound to different units or elements in different modelling techniques, for example,

1. In OO and CBSE, contracts are bound to components, packages, classes, services, or states.
2. In structured programming, contracts are bound to packages, routines, statement blocks [20].

3. In aspect-oriented modelling, contracts are used to specify the link between aspects and classes [13], i.e., weaving.
4. In event-driven approaches, contracts are bound to events and event handlers [17,21].

Fig. 8 outlines a metamodel architecture which is suitable for describing the relationship between contracts and elements in different modelling techniques.

View of Negotiability. The extension of our metamodel in terms of level of negotiability of a contract is based on the layered model in [5]. As shown in Fig. 9, all contract layers in [5] become subclasses of *NegContract*, which represents a negotiable contract. There are two attributes in *NegContract*: *negotiability* and *level*. Negotiability is an in-

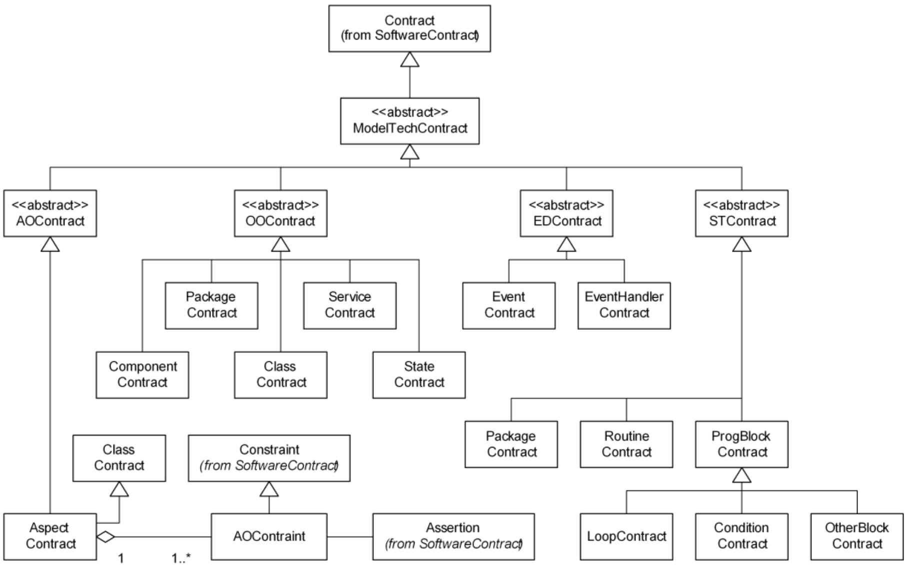


Fig. 8. View of Modelling Techniques

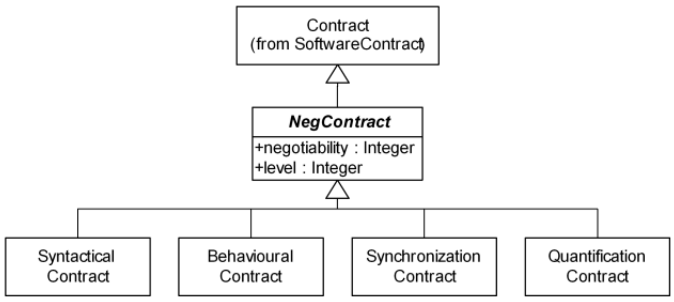


Fig. 9. View of Negotiability

teger value which quantifies the degree of negotiability of a contract: zero represents non-negotiable, while the opposite extreme represents full negotiability. Level is used to preserve the concept of negotiability level in [5], which classifies contracts into four levels of negotiability. There are two ways to preserve prerequisites of contract layers in Beugnard's model: i) multi-tiered inheritance, ii) annotations with OCL. We have chosen the latter approach to avoid complex diagram and to maintain flexibility to changes. The required OCL annotations are in [7].

4 Building a Domain-Specific Contract Language

Domain-specific contract languages (DSCL) have previously been built, often in an ad-hoc manner, to address particular needs and specific domain requirements. DSCLs built in an ad-hoc manner can be difficult to extend and may cause integration and readability problems. They are also unlikely to easily accommodate new requirements in rapidly changing domains, e.g., the Web Services domain.

We now construct a simple DSCL to illustrate the benefit of our systematic extensible metamodelling approach. To continue our metamodelling apporach, the abstract syntax of this simple DSCL is specified with a metamodel, which is closely linked to our software contract metamodel. This will be discussed in section 4.1. We will then show how the metamodel instantiates to become an XML contract language in Section 4.2, thus providing a concrete syntax.

4.1 A Metamodel for a Software Contract Language

When we build a metamodel for a DSCL, we can consider two approaches, either: i) build it from first principles; or ii) extend an existing metamodel. In the former, which generally requires more effort, the language designer has substantial flexibility. In the latter, there may be less flexibility but the language may take less time to design.

Building a full-fledged DSCL metamodel requires a detailed design for its abstract and concrete syntax, as well as formal descriptions of its underlying semantics. Since the focus of our work lies on supporting the process of building DSCLs rather than

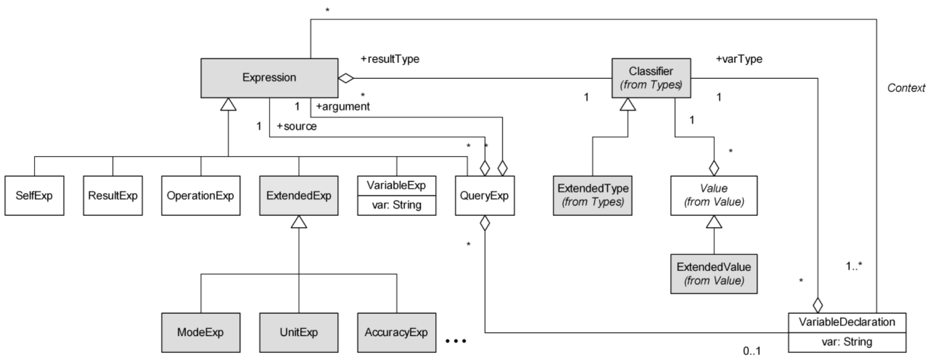


Fig. 10. Metamodel structure for a simple DSCL

producing a full-fledged DSCL, we build our DSCL metamodel based on an existing metamodel. As yet, only OCL and dialects [25] are specified using a metamodelling approach.

We have examined several OCL metamodels, particularly [29,2,25]. We have chosen [29] as the skeleton for our DSCL metamodel due to its completeness and simplicity. Fig. 10 highlights core modifications to the original OCL metamodel. New and modified classes are shaded in the figure. We preserved the package structure and the main class architecture from [29]. In order to integrate with the top-level architecture of our DSCL metamodel, we have made a number of changes to the metamodel:

1. The prefix *Ocl* in all classes is dropped.
2. Controversial classes *OclAny*, *OclType*, and their derivatives (e.g., *OclAnyType*) are removed from the metamodel.
3. The superclass in the *Type* package is changed to *Classifier*. Therefore, every unique classifier in our DSCL can be a type. This allows greater expressiveness at the price of a weaker type system.
4. *ExtendedExpression*, *ExtendedType*, and *ExtendedValue* are introduced in *Expression*, *Type*, and *Value* package respectively. These extension points allow us to attach domain-specific expressions, types, and values.

Recall that we have the notion of *Constraint* and *Mechanism* in our metamodel. The modified OCL metamodel is not a complete DSCL metamodel. As shown in Fig. 11, infrastructure is necessary to 'contain' the modified metamodel. The connection point between this infrastructure and the modified metamodel is in the *Expression* class. In the software contract metamodel, *DomainConstraint* is subclass of *Constraint*. A domain-specific constraint may be implemented by one or more domain-specific expressions. Hence, it is a part of *Constraint* in a *Contract* in the DSCL metamodel. A contract may be used to specify the relationship between two or more parties, i.e., *Server* and *Client*. Various contract enforcement mechanisms were modelled in the software contract metamodel, however, only *ExceptionHandlingPolicy* has a practical meaning in

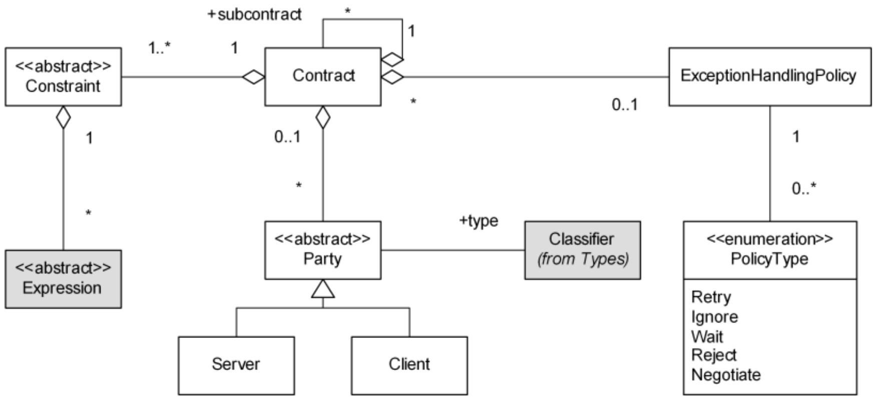


Fig. 11. Metamodel of syntax for a simple DSCL

a DSCL syntax. Therefore, we only preserve *ExceptionHandlingPolicy* from the subclasses of *Mechanism*.

4.2 Building a Contract Language in XML

XML [35] is a popular markup language for information exchange, which is simple, yet highly extensible. We present an example to illustrate how to use the DSCL metamodel to construct an XML-based contract language. Space limitations prevent us from capturing the concrete syntax for the contract language, but doing so is reasonably straightforward. Fig. 12 presents an instantiation of the metamodel for capturing the safety-critical contract in Fig. 1. On the top level, a Contract instance represents the CoolingControl contract. It contains a sub-contract which binds to the poll service withing the component. The poll service contract specifies the constraints for its interface, precondition, and postcondition. Both *AccuracyExp* and *UnitExp* are domain-specific expressions in the safety-critical contract. Unsurprisingly, *AccuracyType* and *UnitType* are the result types of *AccuracyExp* and *UnitExp* respectively.

At this point, the structure of the final XML contract has emerged. We simply rewrite classes in the safety-critical contract model (Fig. 12) into XML constructs, thus producing a concrete instantiation (given a metamodel for the concrete syntax this process could be automated using QVT transformations without difficulty).

1. Classes in the model are converted into XML tags.
2. Roles, i.e., source and argument, become XML tags as well.
3. Class attributes, e.g., *name*, *type*, become XML attributes.
4. Subclass of *Expression* is converted into `<expression>`; its class identifier becomes the type attribute, e.g., *UnitExp* becomes `<expression type='x-unit'>`.

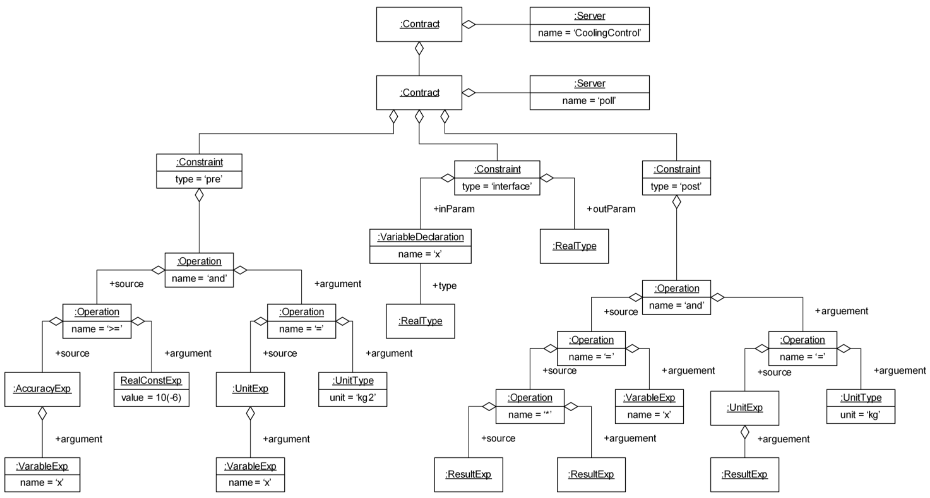


Fig. 12. Instantiation of the metamodel for a simple DSCL

```
<?xml version="1.0"?>
<XCL>
  <contract server="CoolingControl">
    <contract server="poll">
      <constraint type="inteface">
        <InParam type="real" name="x" />
        <OutParam type="real" />
      </constraint>
      <constraint type="pre">
        <expression type="and">
          <source>
            <expression type="greaterequal">
              <source>
                <expression type="x-accuracy">
                  <arguement>
                    <variable name="x" />
                  </arguement>
                </expression>
              </source>
              <argument>
                <const type="real" value="1e-6" />
              </argument>
            </expression>
          </source>
          </argument>
            <expression type="equal">
              <source>
                <expression type="x-unit">
                  <arguement>
                    <variable name="x" />
                  </arguement>
                </expression>
              </source>
              <argument>
                <const type="x-unit" value="kg2" />
              </argument>
            </expression>
          </argument>
        </expression>
      </constraint>
      <constraint type="post">
        <expression type="and">
          <source>
            <expression type="equal">
              <source>
                <expression type="multiply">
                  <source>
                    <expression type="result" />
                  </source>
                  <arugement>
                    <expression type="result" />
                  </arugement>
                </expression>
              </source>
              <arugement>
                <variable name="x" />
              </arugement>
            </expression>
          </source>
          <arugement>
            <expression type="equal">
              <source>
                <expression type="x-unit">
                  <arguement>
                    <expression type="result" />
                  </arguement>
                </expression>
              </source>
              <arugement>
                <const type="x-unit" value="kg" />
              </arugement>
            </expression>
          </arugement>
        </expression>
      </constraint>
    </contract>
  </contract>
</XCL>
```

Fig. 13. Translation of the DSCL metamodel into a contract in XML

5. Arithmetic operations like $'='$ and $'>='$ are converted into expressions since *OperationExp* are subclasses of *Expression* in the DSCL metamodel.

Therefore, the safety-critical contract in Fig. 1 will be represented by a contract in XML in Fig. 13.

5 Conclusion

This paper has preseted a metamodelling approach to the development of domain-specific contract languages. A conceptual metamodel and classification of contracts and views of contracts has been presented, as has a process, CEAP, for refining this metamodel. The metamodel has been used to construct an abstract syntax metamodel for a domain-specific contract language, and this in turn has been instantiated to produce a domain-specific contract language in XML.

The paper was able to provide only an overview of the metamodelling approach. The interested reader should refer to [7] for additional technical details and more comprehensive examples and justifications.

The main aim of the paper from our perspective is to try to convince the reader that a metamodel approach to analysing and designing software contracts and software contract languages is the most suitable way, for building better languages and improve the overall use of contracts. This is mainly achieved via the four viewpoints of the model. The extensible metamodels contribute to software contract language design, which gives a framework for future extensions. This is somehow similar to the idea of allowing plugins in software, and the UML Profiling approach.

Our future work includes instantiation of the metamodelling approach to domain-specific contract languages, particularly for high-integrity real-time systems. We have produced prototype tool support for such language (particularly for use in simulation of real-time systems) and plan to experiment on their use in incrementally certifying these systems as well.

References

1. What is metamodeling, and what is it good for? available at www.metamodel.com, last accessed 5 June 2005.
2. T. Baar and R. Hahnle. An integrated metamodel for OCL types. In *Proceedings of OOPSLA 2000, Workshop Refactoring the UML: In Search of the Core, Minneapolis, Minnesota, USA*, 2000.
3. L. Baresi, G. Denaro, L. Mainetti, and P. Paolini. Assertions to better specify the Amazon bug. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*. ACM Press, July 2002.
4. M. Barnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *Proceedings of the 25th International Conference on Software Engineering*, pages 93–103. IEEE Computer Society, 2003.
5. Antoine Beugnard, Jean-Marc Jézéquel, Noel Plouzeau, and Damien Watkins. Making components contract aware. *IEEE Computer*, 32(7):38–45, July 1999.
6. L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to support fault isolation in object oriented code. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 70–80. ACM Press, 2002.
7. Zhan En Chan. Multiview of a contract metamodel in systems engineering. Msc thesis, Department of Computer Science, University of York, United Kingdom, September 2004.
8. N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In *Proceedings of Policy 2001: Workshop on Policies for Distributed Systems and Networks, LNCS 1995*, pages 18–39. Springer-Verlag, 2001.
9. S. Flake and W. Mueller. An OCL extension for real-time constraints. In *Object Modeling with the OCL, LNCS 2263*, pages 150–171. Springer-Verlag, 2002.
10. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.
11. Microsoft Inc. *Domain-Specific Language (DSL) Tools*, 2004.
12. C. Mingins J.-M. Jézéquel, M. Train. *Design Patterns and Contracts*. Addition-Wesley, 1999.
13. H. Klaeren, E. Pulvermüller, A. Rashid, and A. Speck. Aspect composition applying the design by contract principle. In *GCSE 2000, LNCS 2177*, pages 57–69. Springer-Verlag, 2001.
14. G. Kotonya and Ian Sommerville. *Requirements Engineering: processes and techniques*. John Wiley & Sons, 1998.
15. S. W. Loke and S. Ling. Design-by-contract for Java based mobile agents. In *Proc. OOSDS'99*, 1999.

16. P. Madsen. Enhancing design-by-contract with knowledge about equivalence partitions. *Journal of Object Technology, Special issue: TOOLS USA 2003*, 3(4):5–21, April 2004.
17. A. McNeile and N. Simons. *Methods of Behaviour Modelling: A Commentary on Behaviour Modelling Techniques for MDA*. Metamaxim Ltd., 2004. DRAFT Version 3.
18. Bertrand Meyer. Building bug-free O-O software: An introduction to design by contract, available at www.eiffel.com.
19. Bertrand Meyer. Design-by-contract. Technical report tr-ei-12/co, ISE Inc., 1987.
20. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1997.
21. C. Michiels, M. Snoeck, W. Lemahieu, F. Goethals, and G. Dedene. A layered architecture sustaining model driven and event driven software development, technical report, KU-Leuven.
22. R. Mitchell and J. McKim. *Design by Contract, by Example*. Addison-Wesley, 2001.
23. I. Nunes. Design by contract using meta-assertions. *Journal of Object Technology, Special Edition: TOOLS USA 2002 proceedings*, 1(3):37–56, 2002.
24. I. Nunes. An OCL extension for low-coupling preserving contracts. In *UML 2003, LNCS 2863*, pages 310–324. Springer-Verlag, 2003.
25. Object Management Group. *UML 2.0 OCL Final Adopted Specification*, 2004.
26. Society of Automative Engineers. *Architectural Analysis and Design Language (AADL)*. SAE, 2005.
27. Richard Paige. An encoding of unit and accuracy assertions in UML and OCL, DARP internal report, 2003.
28. A. Radjenovic. *AIM: Architectural Modelling for Managing Change in HIRTS*. DARP HIRTS Project, 2005.
29. M. Richters and M. Gogolla. A metamodel for OCL. In *UML '99, LNCS 1723*, pages 156–171. Springer-Verlag, 1999.
30. David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.
31. G. Rosu and F. Chen. Certifying measurement unit safety policy. In *18th IEEE International Conference on Automated Software Engineering*, page 304. IEEE Computer Society, October 2003.
32. J. Skene, D. D. Lamanna, and W. Emmerich. Precise service level agreements. In *Proc. ICSE'04*. IEEE Press, 2004.
33. Ian Sommerville. *Software Engineering*. Addison-Wesley, 7th edition edition, 2004.
34. R. N. Taylor. Assertions in programming languages. *ACM SIGPLAN Notices,*, 15(1):105–114, 1980.
35. Franois Yergeau, John Cowan, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.1.

Making Metamodels Aware of Concrete Syntax*

Frédéric Fondement and Thomas Baar

École Polytechnique Fédérale de Lausanne (EPFL),
School of Computer and Communication Sciences,
CH-1015 Lausanne, Switzerland
{frederic.fondement, thomas.baar}@epfl.ch

Abstract. Language-centric methodologies, triggered by the success of Domain Specific Languages, rely on precise specifications of modeling languages. While the definition of the abstract syntax is standardized by the 4-layer metamodel architecture of the OMG, most language specifications are held informally for the description of the semantics and the (graphical) concrete syntax. This paper is tackling the problem of specifying the concrete syntax of a language in a formal and non-ambiguous way. We propose to define the concrete syntax by an extension of the already existing metamodel of the abstract syntax, which describes the concepts of the language, with a second layer describing the graphical representation of concepts by visual elements. In addition, an intermediate layer defines how elements of both layers are related to each other. Unlike similar approaches that became the basis of some CASE tools, the intermediate layer is not a pure mapping from abstract to concrete syntax but connects both layers in a flexible, declarative way. We illustrate our approach with a simplified form of statecharts.

Keywords: Metamodeling, MOF, UML, OCL, Concrete Syntax Description, Visual Languages.

1 Introduction

Productivity gains brought by Domain Specific Languages (DSL) [1] have shown the importance of using appropriate modeling languages in the early phases of the software lifecycle. DSLs have triggered the new trend of language-centric methodologies (see [2,3] for first proposals) and are based on the idea that the first step to efficiently treat a problem is to create or to customize a language that allows to describe the problem adequately. The precise definition of DSLs is in practice often a task for domain or methodology specialists who have only basic knowledge on language design. To minimize the effort, all phases of the language definition should be standardized and supported by tools or frameworks.

A modeling language is usually defined in three major steps. The first one is to define concepts of the language, i.e. its vocabulary and taxonomy, as captured by its abstract syntax. Then, its semantics should be described in such a form

* This work was supported by Hasler-Foundation, project DICS-1850.

that the concepts are clearly understood by the users of the language. Finally, it is necessary to precisely describe the notation, as captured by its concrete syntax. Whereas the semantics definition is out of the scope of this paper, we will concentrate on the concrete syntax part, and especially on its relations to the abstract syntax.

The clear separation between abstract and concrete syntax is a technique to cope with the complexity of real-world language definitions since it allows to define the language concepts independently from their representation. For language designers, it is of primary importance to agree on language concepts and on the semantics of these concepts. The graphical representation of the concepts is often considered less important and is described in many language specifications only informally. However, an intuitive graphical representation is crucial for usability and indispensable for tool vendors who want to support a new modeling language with graphical editors, model animators, debuggers, etc. Sometimes, it is appropriate to have for one language more than one graphical representation, for instance when different stakeholders use the same language but need different views on the model. An example of such a language is ORM [4] that provides a graphical syntax intended for ontology engineers and a pseudo-natural syntax intended for non-specialists.

Metamodeling is a widely used technique to capture the abstract syntax of a language. A well defined set of metamodeling constructs such as classes, associations, attributes, etc., complemented with a constraint language such as Object Constraint Language (OCL) allows one to define the concepts of the language and the relationships between them [5]. The abstract syntax is doubtlessly one of the most important parts of language definitions. Each sentence of the language can be represented without loss of semantic information as an instance of the metamodel. Such an instance can be represented in a standardized, textual format based on the general-purpose representation language XMI [6]. Model representations based on XMI are useful for interchanging models between tools but humans need more comprehensible views on models.

Our approach defines the graphical concrete syntax of modeling languages by complementing each metaclass in the metamodel with a *display scheme*. A display scheme contains an iconic and a constraining part. The iconic part introduces a new layer of *display classes* that define the visual objects for the representation of language concepts. The constraining part defines the connection between the instances of the metaclasses and their graphical representation by the instances of the display classes. Technically, the constraining part of a display scheme consists of a *display manager class* that is placed between the metaclass and the display class. Furthermore, display manager classes serve as anchor points for OCL constraints that are used to describe the connection declaratively.

The definition of our representation classes is heavily inspired by visual language definition techniques [7]. Representation classes take for instance into account the spatial relationships between visual objects such as *overlap*, *right*, *hiddenBy*, etc., and whether visual objects are connected by a polyline, curved line, etc. However, there are some noteworthy differences between our approach and common approaches to define a visual language. Firstly, many visual language definitions do not explicitly distinguish between concrete and abstract syntax. In our approach, the classes for the abstract syntax are completely separated from classes for concrete syntax. Secondly, the mainstream approach to define a visual language is by graph grammars (see [8] for an overview). The underlying idea is to generate all syntactically correct sentences of the visual language as derivations of the grammar rules. In order to decide the question, whether or not a given diagram is syntactically correct, a derivation of the graph grammar rules must be constructed. The same question is decided following our approach just by evaluation of constraints attached to the display manager classes.

The rest of the paper is organized as follows. First, in Sect. 2, a simplified version of the statechart language is briefly described. This language will be used as a case study in Sect. 3 where our approach is stepwise developed. Section 4 will give an overview on related approaches. Finally, Sect. 5 will present conclusions and future work directions.

2 The Statechart Language

We briefly introduce here the concepts of a simplified, but yet illustrative version of statecharts [9] whose metamodel is shown in Fig. 1. State vertices might be connected by transitions. A transition has exactly one source vertex and one target vertex. A vertex is either a pseudo state (initial state, choice, etc.) or a state, which is in turn either a composite state (i.e. containing other vertices and transitions), a simple state, or a final state. Transitions are triggered by events. A state machine is given by its top state. Not shown in Fig. 1 are well-formedness

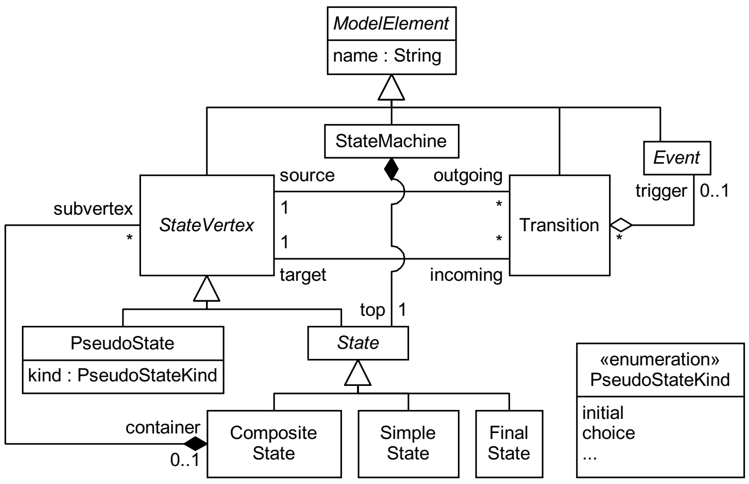
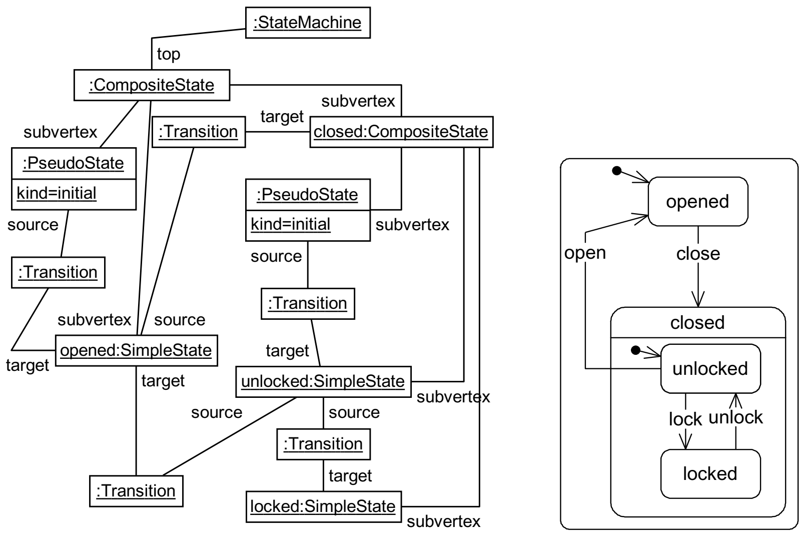


Fig. 1. A simplified metamodel for statecharts

Table 1. Symbols for representation of concepts

Transition	SimpleState	Composite State	FinalState	PseudoState (initial)	PseudoState (choice)
-event->	name	name / contents	◉	●	○



(a) Instantiation of metamodel

(b) More human-friendly, graphical notation

Fig. 2. Two representations of the same statechart

rules that complement the metamodel and stipulate, for example, that a final state can never be the source vertex and an initial pseudo-state can never be the target vertex of any transition.

An informal concrete syntax definition might propose the symbols shown in Table 1 for the representation of language concepts defined in the metamodel. Note that there is no need to define a symbol for `StateMachine` because the state machine is represented by its top state. The intended meaning for the concrete syntax definition is illustrated with Fig. 2. Here, the same statechart sentence is shown both as an raw instance of the metamodel and, in a more intuitive form, using the intended concrete syntax. For the sake of keeping Fig. 2(a) compact, events have been omitted in the metamodel instance.

This simple example reveals already some of the weaknesses of informal concrete syntax definitions. The mapping of concepts to visual objects as given in Table 1 must be complemented by comments stating that the name of a composite state is optional in the upper part of the symbol whereas the lower part optionally shows the representation of the substates. A transition representation (an arrow) starts in one representation of the transition's source and ends in one representation of its target. If a symbol contains parameters as placeholders for additional information, e.g. transitions are supplemented with events, then it must be specified where the information come from, e.g. that the event attached to a transition is indeed the same event that triggers the transition. Another problem is that there may be different icons for the same concept, as for `PseudoState`. Here, it is necessary to describe precisely all conditions for the selection of the correct icon. It is also possible to represent the same concept with variants of the same icon. For instance, a composite state is displayed with or without its name what requires to display or to suppress the name compartment of the symbol.

3 A Scheme-Based Approach to Concrete Syntax Definition

This section presents our approach to define a concrete syntax of a given language. We concentrate here on the definition of a graphical syntax for two reasons. First, most modeling languages provide nowadays an (often informally defined) graphical notation. Second, graphical notations are more challenging as, for example, purely textual notations. In fact, our approach can also be applied for the definition of textual notations. In this case, the display classes on the concrete syntax layer would represent tokens and would be extensions of `String` with additional attributes to encode the location of the currently represented model element.

The concrete syntax is defined by a set of *display schemes*. A display scheme is attached to each metaclass of the metamodel. Although schemes have a formal structure and can be processed by tools, the syntax definition they provide is nevertheless easily accessible by humans.

The scheme-based approach differs in two respects from related approaches. First, we do not aim to define a completely new language but concentrate just on the concrete syntax. This goal is different from what most approaches based on graph grammars aiming at. They define a language from scratch and have to capture in a way both the concrete and the abstract syntax of a language. In most cases, the clear separation between abstract and concrete syntax gets lost. Second, a scheme-based syntax definition intentionally ignores many problems related to tool support for the defined syntax. For example, a graphical editor usually stores the elements of both the abstract syntax layer (the model elements) and the concrete syntax layer (the display objects). The scheme-based syntax definition will provide simple criteria to decide whether or not the model elements are represented correctly by the display objects. However, it is out of the scope of the concrete syntax definition to describe the mechanisms how editors can keep abstract and concrete syntax layers in sync. For example, if the user of the editor creates a new visual object, then, internally, the editor has also to create an instance of the corresponding metaclass and to connect both instances.

3.1 Visual Language Theory

Almost each of todays modeling languages comes with a graphical representation in order to improve readability and usability. Thus, the concrete syntax of modeling languages is usually defined in terms of a visual language. For this reason, we summarize here the relevant basic terms from visual language theory before we explain our approach in detail in the next subsection.

A visual language describes a set of visual sentences which in turn are given by a set of visual elements. A visual element can be seen as an object characterized by values of some attributes. It depends on the language which attributes are important for a graphical element[1], some of the most frequently used attributes are *shape*, *color*, *size*, *position*, *attach regions*.

Visual elements are placed in the Cartesian plane. For some languages, classified in [7] as *geometric-based languages*, the position of visual elements is an important information. Other languages ignore the position of elements but focus on the connections between them (*connection-based languages*). In fact, most real-world languages show characteristics of both geometric-based and connection-based languages and are thus called *hybrid languages*. The strong classification into geometric-based and connection-based languages is notwithstanding extremely helpful since it uncovers the 'ingredients' a visual language can have.

For geometric-based languages there are two possibilities to encode the position of a visual element. If the language is based on *absolute positions* then a sentence consisting of a circle and a square placed at point (1,0) and (2,0), respectively, is different from the sentence where the circle is placed at (1,0) and the square is placed at (3,0). If the language is based on relative positions (spatial relationships) then both sentences would be described by the fact that the square is placed to the right of the circle. Some of the most frequently used spatial relationships are *right*, *up*, *contain*, *overlap* (see [7] for a more complete list). It heavily depends on the visual language which of the spatial relationships are considered to be important. Sometimes, languages are geometric-based even if it seems that the visual elements can be arranged freely. One example is the language of UML class diagrams. At a first glance, rectangles for classes can be placed freely at any point in the space. For instance, a diagram consisting of two rectangles labeled with `A` and `B` would always be read as the same sentence no matter where the (rectangles for) class `A` and `B` are placed. However, there is one exception from this rule: If - let's say - the rectangle for `B` appears completely inside the rectangle for `A`, then the class `A` is read to be composed of class `B`. Thus, the spatial relation *contain* is important to define the visual representation of class diagrams whereas the relations *right*, *up*, etc., do not play any role here.

Connection-based languages allow visual elements to be placed arbitrarily in the space. None of the spatial relationships has an influence on the parsing of

[1] There is a common classification of attributes into graphical, syntactical and semantic attributes. Only the first two classes of attributes are relevant for our approach because semantic attributes are already captured by the abstract syntax definition.

a sentence of such languages. Instead, it is an important information whether two elements are connected by a connector (usually a line, polyline, curved line) or not. Connectors start and end in special regions of visual elements, so-called *attach regions*. A visual object can have one or more attach regions which sometimes collapse to attach points. As already mentioned, visual language definitions formalize an attach region of a visual element just as an attribute of it. This abstracts from the problem to define where an attach region is exactly located in respect of the visual element (e.g. in the lower right corner). However, some symbol editors, e.g. VLDESK [10] or AToM^3 [11], allow to exactly define the position of attach regions inside a visual element. They also solve the very similar problem of defining a shape for visual elements.

3.2 Scheme-Based Definition of Concrete Syntax

The definition of a concrete syntax means to define (1) a visual language, i.e. visual elements with relevant attributes and relationships between them and (2) how the visual elements are connected to the concepts of the language they are supposed to represent. Figure 3 gives an overview how both goals are basically achieved by our approach: A sentence of a visual language, i.e. a set of visual objects, is first mapped to a set of display objects. This mapping and the formalism to define the graphical rendering of visual objects is intentionally left open in our approach. We have experienced with Scalable Vector Graphics (SVG) [12], a language to describe diagrams, but other formalisms or existing tools as symbol editors can be applied for this purpose as well.

Display classes declare for display objects attributes and operations what helps to lift up the abstraction level on which the syntax definition is given. An attribute of a display object summarizes the value of more low level attributes of the underlying visual object such as *xpos*, *ypos*, *size*, *shape*, *color*, etc. The operations of a display object – as we will see later, operations correspond to spatial relationships such as *contain*, *overlap*, etc. – have to be implemented by the underlying visual object. If SVG is taken as a formalism to describe visual objects, the implementation can be done smoothly. If another formalism is taken, some additional adapter classes might be required.

The connection between display objects and model elements is given by display managers which are attached to model elements. Usually, each metaclass is

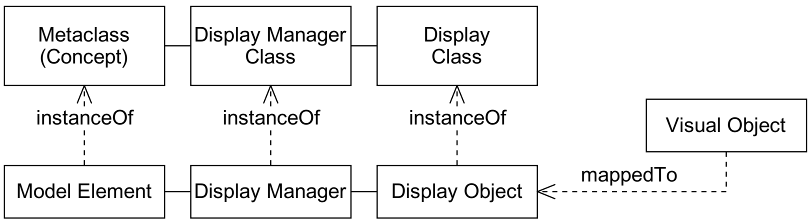


Fig. 3. Scheme definition architecture

connected with exactly one display manager class that in turn is connected with the display class defining the graphical representation. The criteria for a syntactically correct representation are defined in form of OCL invariants attached to the display manager classes. A set of display objects is a syntactically correct representation of a model, i.e. a set of model elements, if and only if the display managers attached to the model elements satisfy all invariants of the display manager classes.

3.3 A Concrete Syntax Definition for Statecharts

We illustrate our approach with a formal definition of the concrete syntax of statecharts whose abstract syntax was given in Sect. 2. Prior to the formal definition, an informal version of it should illustrate the gap between the abstract and concrete syntax, as already introduced in Sect. 2:

Problem 1. A text is shown on the top of transitions to represent the triggering event if it exists;
Problem 2. Depending on the viewer's choice, a composite state is depicted either by a text showing the name of the composite state, or by a region showing the contents of the composite state (i.e. its contained states), or both. In the latter case, the two regions are separated by a line;
Problem 3. The plain side of the transition icon is connected to a representation of its source state; the arrow side is connected to a representation of its target state;
Problem 4. The shape of a pseudo-state representation depends on its kind.

Figure 4 shows the backbone of the statechart concrete syntax definition. Four display schemes for graphically representable concepts of statecharts are defined: `Transition`, `SimpleState`, `CompositeState`, and `PseudoState`. The display scheme for `FinalState` has been omitted for the sake of brevity. All other concepts defined in the metamodel are either abstract (`ModelElement`, `State`) and thus will be depicted by the scheme of their subclass, or are displayed implicitly by the concepts they are attached to (`StateMachine` by its top state and `Event` by the transition it triggers). Note that the missing display scheme for `Event` might make the graphical representation of a model incomplete. If an event does not trigger any transition (according to the metamodel it is not mandatory to trigger at least on transition) then this event is not shown in the representation.

Each display scheme can be split into two parts. The iconic part defines the graphical rendering of visual objects. The constraining part fills the gap between the model elements and the display objects. A display manager class is connected to exactly one metaclass by the association end `me` (for model element); thus, every display manager refers to exactly one model element. The cardinality of the opposite association end `dm` (for display manager) encodes how many different display managers can exist for each model element. Following the syntax definition given in Fig. 4, model elements of `Transition` and `PseudoState` can only be depicted at most once whereas instances of `SimpleState` and `CompositeState`

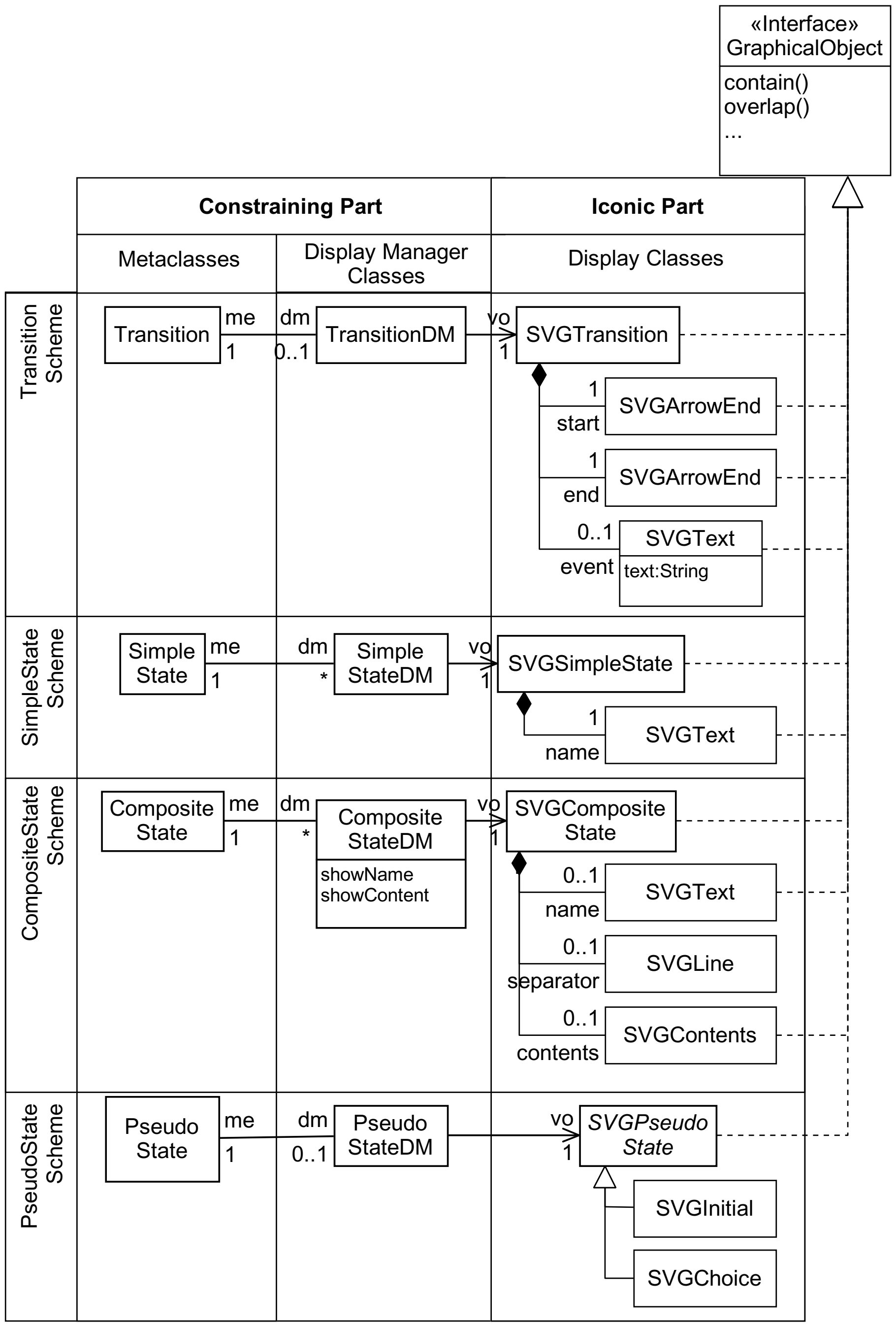


Fig. 4. Display schemes for statechart metaclasses

may be represented arbitrarily often. Thus, also such representations of a statechart are syntactically correct that omit parts of the model or show some states more than once.

Iconic Part of a Scheme. For each display manager class there is always a standard association to the corresponding display class with multiplicity 1 and role name `vo` (abbreviation for visual object) on the end of the display class. A display class represents an abstraction of visual objects that have to be defined in terms of *shape*, *color*, etc. It also declares some query facilities. Some standard queries, as introduced in Sect. 3.1, are declared in interface `GraphicalObject` that must be implemented by every display class. This ensures that any display object is capable to respond to such queries. As seen below, queries are heavily used in the OCL invariants attached to display manager classes.

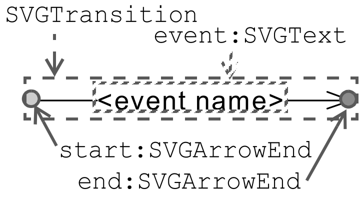


Fig. 5. The icon for `Transition`

Often, a model element is not displayed just by one atomic visual object but rather by a composition of such objects. Thus, the *main display object* linked to the display manager is composed of sub-objects whose position, size, etc., is controlled by the main display object. Figure 5 illustrates the internal definition of `SVGTransition`, the display class to represent transitions. Objects of `SVGTransition` are composed of one sub-object to display the event and two sub-objects representing attach points. Whereas the sub-object to display the event is optional, the two sub-objects of type `SVGArrowEnd` are mandatory.

Besides composing display classes, it is also sometimes necessary to subclass them. The class `SVGPseudoState` is an example. The concept `PseudoState` is represented depending on the value of attribute `kind` by completely different shapes. Each of these shapes is defined by a single display class (e.g. `SVGInitial` for initial states, `SVGChoice` for choices) that inherits from `SVGPseudoState`. The class `SVGPseudoState` itself is declared as abstract.

Constraining Part of a Scheme. Based on the backbone shown in Fig. 4, the relationship between abstract and concrete syntax layers can be formalized by OCL invariants. In order to illustrate the expressive power of these formal constraints, we discuss now each of the four, already sketched problems of informal syntax definitions.

Problem 1 requires to keep attribute values for model elements and representing display objects in sync. This problem can be resolved by the following constraint:

```
-- Problem 1
   context TransitionDM
   inv: if self.me.trigger->isEmpty()
        then self.vo.event->isEmpty()
        else self.vo.event.text = self.me.trigger.name
        endif
```

The constraint ensures that the correct name of the event is displayed whenever a transition is triggered by an event. Sometimes, it might be appropriate to relax this rule so that the triggering event of a transition can be suppressed in the graphical representation. In this case, the invariant looks as follows:

```
-- Problem 1 with optional event display
   context TransitionDM
   inv: self.vo.event->notEmpty() implies
        self.vo.event.text = self.me.trigger.name
```

Problem 2 is an example for user-directed representation policies which can be encoded by attributes of type `Boolean` in the display manager class. Problem 2 is captured by the following constraint:

```
-- Problem 2
   context CompositeStateDM
   inv: self.showName = self.vo.name->notEmpty() and
        self.showContent = self.vo.contents->notEmpty() and
        (self.showName and self.showContent)
       = self.vo.separator->notEmpty()
```

Problem 3 is similar to Problem 1 but the synchronization between model and representation cannot be achieved by constraining the values of attributes. Instead, the spatial relationships between display objects have to be taken into account. This can be done by an OCL invariant due to the declaration of `overlap(GraphicalObject):Boolean` in interface `GraphicalObject` that is implemented by all display classes. Note that the semantics of the query `overlap` is hidden in the implementation of the visual objects. Thus, the 'correctness' of the OCL invariant depends on the 'correctness' of the implementation of `overlap` in the visual objects.

```
-- Problem 3
   context TransitionDM
   inv: self.me.source.dm.vo->one(svo |
                          self.vo.start.overlap(svo)) and
        self.me.target.dm.vo->one(tvo |
                          self.vo.end.overlap(tvo))
```

Problem 4 is an example for the representation of modeling elements belonging to the same concept (`PseudoState`) by different shapes. The OCL invariant takes advantage of OCL's ability to check the actual type of an expression with `oclIsKindOf()`.

```
-- Problem 4
   context PseudoStateDM
```

```
inv: let kindAsso:Set(TupleType(kind:PseudoStateKind,
                                type:OclType)) =
          Set{Tuple{kind = PseudoStateKind::initial,
                    type = SVGInitial},
              Tuple{kind = PseudoStateKind::choice,
                    type = SVGChoice}}
     in
        self.vo.oclIsKindOf(
          kindAsso->any(t| t.kind = self.me.kind).type)
```

4 Related Work

The problem of defining a graphical concrete syntax on the top of a metamodel has already been addressed by the OMG and numerous authors.

The OMG has adopted a standard for diagram interchange for UML2.0 (UML-DI [13]) to overcome the shortcomings of model interchange based on XMI. Indeed, XMI focuses on transmitting pure modeling data, given by the abstract syntax of models, and ignores graphical information. UML-DI provides a generic metamodel for extending any other metamodel so that graphical information can also become part of the data interchanged in XMI. However, UML-DI only concentrates on gathering graphical data and does not focus on how those data are structured. Consequently, these data are still ambiguous and tools interchanging them still need to agree on their meaning. For instance, the UML-DI meaning for UML is defined using an XMI to SVG translator that cannot be reused for a completely new language. Moreover, neither UML-DI nor XMI-to-SVG translators capture spacial relationships in order to express, for example, that two graphical elements do overlap.

Other approaches, like XMF [14], argue that the concrete syntax involves a representation language. An example of such languages is Scalable Vector Graphics (SVG) [12], but the XMF framework provides its own graphical language in form of a representation metamodel with well defined semantics. Here, semantics corresponds to a graphical representation rendering. Thus, to define the representation of a given language whose abstract syntax is given by a metamodel, it is sufficient to define a model transformation between the metamodel of the language and the representation metamodel.

Another approach is taken by most meta-CASE tools, like GME [15], DOME [16], MetaCASE [17], or AToM^3 [11]. In principle, they define a representation template for each metaclass in the abstract syntax. A template includes a set of representation language constructs, as instances of the representation language metamodel, together with some holes to make variants in the representation possible. Again, each of these tools impose its own graphical language. When a model element has to be represented, the holes are replaced depending on relevant information from the current model. Unfortunately, while most of these tools provide a constraint language that can be used to impose restrictions on the abstract syntax, they do not provide access to the concrete syntax. A notable exception is AToM^3 that allows constraints written in Python to select among variations of the icons. However, also in AToM^3 the definition of the concrete syntax is done at a much lower level as our approach which uses OCL as the main language to specify the concrete syntax.

Graph-grammar based language definitions (as Triple-Graph-Grammar [18], GenGED [19]) are constructive and aim at finding a derivation for a given diagram. In addition to rules, GenGED offers the possibility to attach constraints to the concrete syntax classes (called type graph nodes in the GenGED terminology), but the purpose of the constraints is merely the computation of a possible layout for a diagram. The language definition itself is still based on graph grammar rules (see [20] for the GenGED definition of the same statechart fragment as we used here for illustration).

5 Conclusion

We have presented a way to specify the concrete syntax of languages whose abstract syntax is already available in form of a metamodel. The main idea is to complement the metamodel with display schemes.

The iconic part of a scheme defines some display classes for representing model elements. Variants in the representation are expressed by attributes or methods attached to the display classes. We do not impose any language to define display classes but assume that display classes do implement the interface `GraphicalObject`.

The constraining part of a scheme consists of a display manager class together with associations to metaclasses and display classes as well as a number of constraints. The purpose of the constraints is to stipulate restrictions for the visualization of model elements. The expressive power of constraints has been illustrated by applying them on a simplified version of the statechart language.

Our scheme-based approach resides at a higher level abstraction than most other approaches. Except for the shape information for the icons, only OCL constraints have to be attached to display manager classes. The relatively high number of new classes that must be defined is outweighed by the fact that many of these classes can be defined mechanically.

We are currently implementing our approach in form of a free editor that is customizable with modeling language specifications. The user is able to place different symbols on a canvas and by creating a symbol (instance of a display class) the corresponding display manager and model element is created as well. At any time, the user can ask the editor whether the current diagram is syntactically correct or not. Internally, the editor evaluates then all OCL constraints attached to the extended metamodel. This check might be costly if implemented naively because each constraint must be checked for a rather high number of objects. A solution for this problem is to use strategies to determine only those constraints that might be broken by the changes in the diagram made after the last check (see [21]). However, the efficiency aspect becomes less important if our free editor is seen as a reference implementation for concrete syntax definitions.

Other tool vendors, that have implemented the same language using more efficient techniques, could test whether their tools comply with the formally given syntax of the modeling language or not.

References

1. Richard B. Kieburtz, Laura McKinney, Jeffrey M. Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino Oliva, Tim Sheard, Ira Smith, and Lisa Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, pages 542–552, 1996.
2. Stuart Kent. Model driven engineering. In Michael J. Butler, Luigia Petre, and Kaisa Sere, editors, *Proceedings of Third International Conference on Integrated Formal Methods (IFM 2002)*, volume 2335 of *LNCS*, pages 286–298. Springer, 2002.
3. Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Guest editors' introduction: Model-driven development. *IEEE Software*, 20(5):14–18, 2003.
4. Terry Halpin. *Information Modeling and Relational Databases : From Conceptual Analysis to Logical Design*. Morgan Kaufmann, second edition, 2003.
5. OMG. Meta-Object Facility (MOF) 1.4. OMG Document formal/02-04-03, April 2002.
6. OMG. XML Metadata Interchange (XMI) 2.0. OMG Document formal/03-05-02, May 2003.
7. Gennaro Costagliola, Andrea De Lucia, Sergio Orefice, and Giuseppe Polese. A classification framework to support the design of visual languages. *Journal of Visual Languages and Computing*, 13(6):573–600, 2002.
8. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
9. David Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
10. Gennaro Costagliola, Vincenzo Deufemia, and Giuseppe Polese. A framework for modeling and implementing visual notations with applications to software engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 13(4):431–487, 2004.
11. Juan de Lara and Hans Vangheluwe. Using AToM^3 as a meta-case tool. In *Proceedings of the 4th International Conference on Enterprise Information Systems (ICEIS)*, pages 642–649, 2002.
12. W3. Scalable Vector Graphics (SVG) 1.1 Specification, January 2003.
13. OMG. UML 2.0 diagram interchange specification - final adopted specification. OMG Document ptc/03-09-01, September 2003.
14. Tony Clark, Andy Evans, Paul Sammut, and James Willans. Applied metamodelling: A foundation for language-driven development. Available at http://albini.xactium.com, 2005.
15. Matthew J. Emerson, Janos Sztipanovits, and Ted Bapty. A MOF-based metamodeling environment. *Journal of Universal Computer Science*, 10(10):1357–1382, 2004.
16. Honeywell. Dome users guide. http://www.htc.honeywell.com/dome/support.htm, 2000.
17. MetaCase. Abc to metacase technology. http://www.metacase.com/papers, 2004. White Paper.

18. Andy Schürr. Specification of graph translators with triple graph grammars. In *Proceedings of 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'94)*, volume 903 of *LNCS*, pages 151–163. Springer, 1995.
19. GenGED Team. GenGED homepage. http://tfs.cs.tu-berlin.de/~genged/, 2005.
20. Roswitha Bardohl, Hartmut Ehrig, Juan de Lara, and Gabriele Taentzer. Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation. In *Proceedings of 7th International Conference on Fundamental Approaches to Software Engineering (FASE 2004)*, volume 2984 of *LNCS*, pages 214–228. Springer, 2004.
21. Jordi Cabot and Ernest Teniente. Determining the structural events that may violate an integrity constraint. In Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors, *Proceedings of UML 2004 - The Unified Modelling Language: Modelling Languages and Applications*, volume 3273 of *LNCS*, pages 320–334. Springer, 2004.

XRound: Bidirectional Transformations and Unifications Via a Reversible Template Language

Howard Chivers and Richard F. Paige

Department of Computer Science, University of York, York, YO10 5DD, UK
Fax: +44 1904 432767
hrchivers@iee.org, paige@cs.york.ac.uk

Abstract. Efficient tool support for transformations is a key requirement for the industrialisation of MDA. While there is substantial and growing support for unidirectional transformations (e.g., from PIM-to-PSM), for bidirectional transformations there is little. This paper presents tool support for bidirectional transformations, in the form of a language, called XRound, for specifying *reversible templates*. The language supports round-trip transformations between UML models and predicate logic. Its supporting tool also implements *model unification*, so that new information encoded in logic can be seamlessly integrated with information encoded in the model.

1 Introduction

Transformations are a critical component of Model-Driven Development, particularly in the MDA [2]. To this end, the Queries-Views-Transformations (QVT) [1] standard has been developed, in order to provide a precise, flexible mechanism for modelling transformations between models. Even though QVT is still in the process of standardisation, several tools and QVT-compatible (or QVT-like) languages have been developed for supporting the transformation process. Of note amongst these are QVTMerge [3] and the Atlas Transformation Language (ATL) [4], the latter of which provides substantial tool support for model transformation; similarly, XMF [5] provides modelling support for transformations based on an executable dialect of OCL. There are also transformation tools outside of arena of OMG standards; for example, the TXL [7] framework has some similarities to QVT, though it has been predominantly targeted at programming language transformation. The generative programming community has made use of templates to accomplish similar tasks [6], and the meta-programming language Converge [9] has been successfully used to implement a transformation language as an instance of a domain-specific language.

QVT transformations can be *unidirectional* (i.e., from one metamodel to a second, not necessarily new, metamodel) or *bidirectional* (i.e., reversible between two metamodels). The former is of critical use in MDA, e.g., for transforming platform independent models (PIMs) into platform specific models (PSMs). The latter is vital for supporting *rigorous analysis* of models: the results of analysis may need to be reflected in the source of a transformation. For example, a static analysis may be applied to a PSM, resulting in changes being made to that PSM. These changes may need to be reflected in the original PIM.

Limited tool support currently exists for bidirectional transformations; it can be partly supported using sequential application of unidirectional transformations, but this is not entirely satisfactory because information – e.g., diagram layout, detailed representations of platforms – is likely to be lost after each unidirectional transformation is applied.

Related to model transformation technology is *model composition* technology; this is sometimes also referred to as model *merging*, *weaving*, or *unification.* With model composition, two or more models (usually of parts of the same system) are combined into one, in the process resolving inconsistencies, overlaps, and nondeterminism. As of yet, there is minimal language and tool support for model composition; the Atlas Model Weaver [8] is one of the first generic prototypes. Model composition techniques could alleviate some of the problems with using unidirectional transformations for supporting round-trip engineering.

This paper presents a new, template language, called *XRound*, for supporting bidirectional transformations. This language is not QVT-compatible, as of yet, but it uses standard underpinning technology and mechanisms that suggests it could easily be made so. Moreover, the paper presents powerful tool support for this language that allows bidirectional transformations, as well as a form of model unification.

1.1 Context and Contribution

The template language described in this paper arose from the need to support round trip engineering from a specialized analytic tool. The tool, the *Security Analyst Workbench (SAW)*, carries out risk-based security analysis of UML system models, and provides an environment in which the user can interactively set and test security policies. The resulting policies (e.g., access controls) are part of the system design, so they must be re-integrated into the engineering documentation, i.e., the UML models.

SAW does not need the whole of a UML system model on which to perform risk-based analysis; it needs a view that describes certain features of the system (such as classes, operations, and stereotypes), and these are expressed as predicates. For example, *(class,foo)* would identify a class object named *foo*. Predicate representation is the basis of the model unification that we describe in the sequel.

Originally, SAW used *xsl* templates to generate a predicate view from an XMI representation of a UML model. Template processing provided a bridge between tool-specific XMI and the analysis application, which remained UML-tool independent. This allows designers to use their preferred UML environment, and is preferable from the tool software perspective; for example, type checking of security properties is implemented once within the analytic tool, rather than in each UML environment.

Template processing provides an important bridge between different tools, but the available solutions are unable to support the reverse path of unifying the output data back into its original source. Round-trip engineering of analysis results back into the UML is therefore not straightforward with a conventional template processor, but is a significant requirement for specialist analytic tools.

XRound is designed to overcome this problem. Its objective is to maintain the advantages of template processing, including simple scripting of data transformations and independence between input and output applications, while supporting bi-directional transfers, and unification, of data. This language and its supporting processor allows the SAW application to import UML designs in tool-specific XMI, and re-generates the XMI when the analytic model is changed.

The contribution of this paper is that it describes a new template language with the unique ability to support transformations in both directions, the general principles that underline its design, and a template processor for the new language.

This paper continues by describing XMISource, which is a Java-based processor for XRound. The processor is presented first to clarify how the language will be used, including its straightforward client application interface. The principles behind reversible templates are then introduced, and the structure of the template processing is described. The template language is then presented in detail; the core syntax is first described, followed by two worked examples. Further sections describe language features that support performance management and debugging, and summarise limitations in the current implementation.

2 The Template Processor

The purpose of this section is to clarify the system in which the template language will be used. The first implementation of an XRound processor is a Java class that encapsulates an XML file and allows its client application to important and export predicates from and to the XML source. The design of the processor is given in Fig. 1. Although this is named XMISource after its main application, there is nothing XMI-specific in XRound or in this processor.

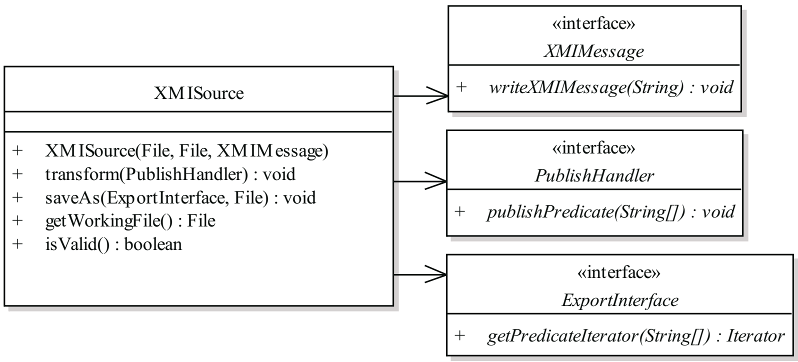

«interface»
XMIMessage
+ writeXMIMessage(String) : void
XMISource
+ XMISource(File, File, XMIMessage)
+ transform(PublishHandler) : void
+ saveAs(ExportInterface, File) : void
+ getWorkingFile() : File
+ isValid() : boolean
«interface»
PublishHandler
+ publishPredicate(String[]) : void
«interface»
ExportInterface
+ getPredicateIterator(String[]) : Iterator

Fig. 1. The XRound Template Processor

The XRound processor has a single class, XMISource, which encapsulates an XML file whose name is provided in the constructor. Three interfaces are defined in the package, and these call-backs are provided by the application client to allow the processor to import and export predicates.

Predicates are represented as arrays of Strings, such as {class,foo}, which describe features in the XML input that are required by the application. The processor supports three transform operations: *validation*, *import* and *export*. (see Template Processing for further detail).

Validation. The XMISource constructor takes three parameters, the reference XML File, the Template File, and a message interface. (The Java File class encapsulates a file name.) The message interface is used to pass certain error messages back to the application, particularly those that report inconsistencies between the template and the XML input. A message interface is used in preference to a thrown exception, since it allows a sequence of messages to be reported during processing, which is valuable during template debugging.

The initialization process parses both the Template and XML input file, and executes a section of the template which is intended to validated the input. Methods are provided to allow the client application to check that the validation was successful (isValid) and to retrieve the name of the XML input file (getWorkingFile).

Import. A single method, transform, runs the import process, which extracts predicates from the XML input, as specified by the template, and publishes them to the client application. As each predicate is constructed the PublishHandler interface provided by the application client is called to transfer the predicate to the client.

Export. A single output method (saveAs) is provided to export predicates from the client application to a named XML file. The output filename is provided by the client, together with an interface (ExportInterface) which allows XMISource to obtain predicates from the application. This is slightly more functional than the other interfaces, but is still straightforward: the client is provided with an incomplete predicate, which is an array of Strings, some elements of which may be null. The client responds with an iterator, which encapsulates predicates matching this template.

The export function updates the reference XML input with predicates obtained from the application, and then writes the result to the named File. File naming strategies and backup files, etc, are implemented by the client application.

Because the input XML is retained, there is no need for the complete XML tree to be exported to the application; the transformation therefore includes only the features required by the application.

The important feature of the template processor is its straightforward client interface; this is a direct result of the reversible template model, since:

- The application only needs to obtain the predicates that it needs for its function, the rest of the input XML remains hidden.
- The application interface is independent of the tool used to generate the XML: any tool differences are accounted for in the template.
- The template includes an explicit validation section that is run at initialisation.

The internal design of the processor is beyond the scope of this paper, but an outline of how the three main operations relate to the template specification is discussed in the next section, before the language itself is given.

3 Template Processing Overview

This section introduces the key concepts behind a reversible template, then describes how the need for the processing operations described above motivate the coarse structure of the template language.

3.1 Bidirectional Transformations and Model Unification

Template processing is usually a one-way operation as shown in figure 2: the template processor locates elements in the input tree and publishes them, suitably formatted.

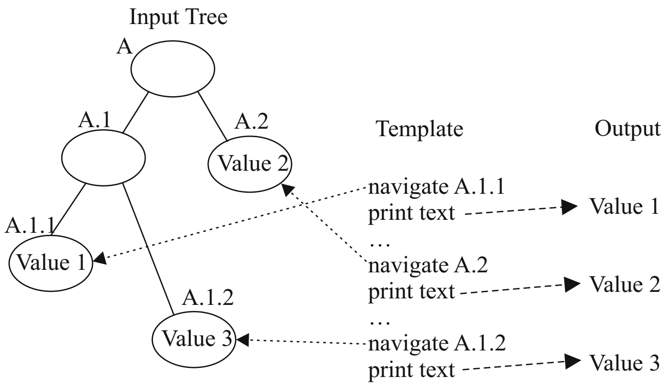


Fig. 2. Conventional Template Processing

In the case of XML data, such as XMI, the input to the template processor is a tree; the output may be XML, or it may be published in another format such as text or HTML. Conventional templates are capable of encapsulating comprehensive programming behaviour, but their fundamental structure is still to navigate to selected nodes in the input tree, extract information, and produce suitably formatted output. The benefit of a template over a standard programming language is usually that it is tailored to the particular type of input and output required.

Reversible templates defined in XRound are similar in structure to existing templates, but encapsulate a fundamentally different type of operation: unification. The operation of a reversible template is shown in Fig. 3.

A reversible template navigates to elements in the input tree, in a similar way to a conventional template, but it also references values in the application predicate. The fundamental operation is to match, or *unify*, values in the source tree with values in the predicate. Unification allows values to be determined from either the source tree, or the application predicate, or if values are set in both, to ensure that they are consistent. For example, in Fig. 3 the first value is not known in the source, but is available in a predicate; the opposite is true for the second value; and the third is the same in both source and predicate, so this unification succeeds.

This underlying unification process determines the design of the template language; as well as carrying navigation information to identify information in the source tree, each part of the template identifies a unification slot, and the fundamental operation is ‘match’, which is to unify the slot with either the XML input tree, or the application predicate.

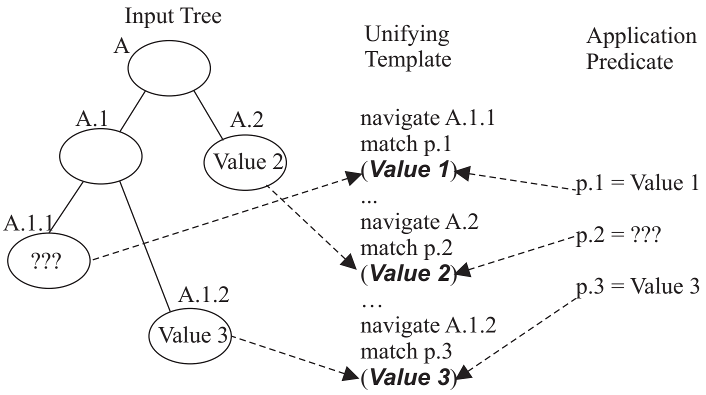


Fig. 3. The Template Unification Process

Unification is conceptually straightforward, but designing a template language that exploits this process does present some problems, including:

- The source navigation for a reversible processor is not quite the same as a conventional template processor, because is has to unify input nodes that do not exist. For example, in Fig. 3 it is not simply the case that the input node does not have the first value set, but that the whole node (A.1.1) is missing. The template language must allow the programmer to specify which nodes are allowed to be missing, and which areas in the source tree are fixed. In XRound, nodes that may be missing are marked as *mutable* and can also be created by the template processor during the process of reverse engineering.
- Because some nodes in the input tree may be missing, it is not straightforward to select nodes based on an attribute value, as is possible in an Xpath expression. In XRound this problem is solved by a general constraint mechanism, which constrains unification slots to specified values. Constraints are also unified as part of the matching process and can therefore be used to specify the types of predicate that can be generated, constrain XML node selection, and determine application predicates to be unified.

The underlying unification process determines some features that are needed in a reversible template language: the definition of unification slots and slot constraints. The next section describes how the main operations of the template processor are supported.

3.2 Template Processing

This section describes the operation of template processing in sufficient detail to introduce the clause structure of the template language.

The previous section described the process of unification, and this places some requirements on the sections, or clauses, of the template language. Essentially a clause must:

- Specify a number of unification slots.
- Allow the specification of constraining values for each slot.
- Unify values in the XML input and/or in application predicates with slot values and constraints.

In order to allow for a separate verification section, and also to allow the user to distinguish parts of the XML input that should be fixed, as opposed to those that may be rewritten, three types of clause are defined in XRound:

- *validate*
- *structure*
- *roundtrip*

A *validate* clause specifies validation checks, a *structure* clause references elements of the XML input that should not be modified, and a *roundtrip* clause includes input nodes that may be modified when the XML is regenerated from application predicates. The value of the *structure* clause is that it allows a wider range of navigation types and some performance optimisations compared to *roundtrip* clauses, because it does not have to account for missing nodes. However, it is not the case that all nodes visited by *roundtrip* clauses can or should be re-written; nodes that can be updated are specifically identified in XRound by a *mutable* attribute.

The three main processing operations can now be described:

Validation. Validation can be used to make any checks that the programmer requires, but its primary aim is to ensure that the template and XML input are compatible. Because XMI is tool specific, a particular template will apply to a limited range of tools and versions; validation clauses in the template are used to check that the input data (e.g., tool type and version number) are compatible with the current template.

After the XMI input and the template have been successfully opened and parsed, each *validate* clause is executed, and each must succeed for the validation to succeed. No other clauses are executed during validation, and the validation clauses are not executed as part of any other processing.

Import. The import operation is similar to normal template processing, it is used to assemble predicates from the XML input and provide them to the client application.

Any *structure* clauses are first executed, followed by *roundtrip* clauses. Each clause is unified with constraints specified within the clause, but not with any application predicates. The clauses have one or more *publish* attributes that mark completion; when these are reached the unification slots within the clause are checked and, if complete, a predicate is exported to the client.

Export (saveAs in XMISource, see Fig. 1). The export operation merges predicates from the client application back into the XML input, then saves the result. The purpose of the operation is to update the XML representation with any changes that have been made by the application, without the need for the application to manage the specific XML format, and without the need to write different templates for input and output processing.

The first processing stage executes all the *structure* clauses in the template; although this will not result in any updates to the XML output, it is necessary because it may build reference information that is used later (see Performance Management, below). There are two further processing stages, the second removes mutable nodes, assuming that nodes no longer present in the application have been deleted intentionally, and the third re-builds nodes from the application predicates. In both cases, the operation (remove, build) takes place only for mutable nodes that have been encountered during a successful unification of a roundtrip template clause. The values written to the rebuilt nodes are obtained from the unification slots in the template, and so may contain values from the application predicates, from the XML input, or directly from clause constraints. The relevant application predicate is obtained when the clause is encountered; essentially, the template processor builds a predicate mask that matches any fixed values specified in the constraint clauses, and requests the application for an iterator over all predicates that match the mask. The clause is then executed once for each predicate in the iterator.

In summary, the process that allows a template to be interpreted in both directions is unification; this has implications for the types of navigation that can be carried out within a template and determines the need for other structure in each clause: unification slots and constraints. The three key operations of validation, import and export are supported by the clause structure in XRound, allowing the programmer to specify validation checks (*validate*), elements of the XML that should not change (*structure*), and parts of the XML tree that can be modified (*roundtrip*).

4 The XRound Language

This section describes the XRound language. It begins by describing how an XRound template is organised in terms of clauses and how they support unification slots, constraints, and transformations. This is followed by a detailed description of transformations, and two examples of template clauses. This section concludes by describing language features that support performance management and debugging.

4.1 Basic Template Structure

The top-level structure is most easily described with an abbreviated DTD:

```
<!ELEMENT tpl.template
((tpl.validate|tpl.structure|tpl.roundtrip)*)>

<!ELEMENT tpl.validate (tpl.constraint*,tpl.specification+)>
```

```
<!ATTLIST tpl.validate  length CDATA #IMPLIED
                        auxLength CDATA  #IMPLIED>

<!ELEMENT tpl.stucture (tpl.constraint*,tpl.specification+)>
...

<!ELEMENT tpl.roundtrip (tpl.constraint*,tpl.specification+)>
...

<!ELEMENT tpl.constraint (tpl.value+)>
<!ATTLIST tpl.constraint position CDATA #REQUIRED>
<!ELEMENT tpl.value (#PCDATA)>
```

A template is a well-formed XML document containing three node types that may occur in any number and any order: *tpl.validate, tpl.structure* and *tpl.roundtrip.* These are the *clauses* introduced in the previous section. Attributes in each clause node specify the number of unification slots (*length* + *auxLength*) and these are simply indexed as an array in the subsequent template (e.g. *position* = *"0"*). The slots are divided into two, and the first section (specified by *length*) is mapped directly to an application predicate.

Each clause may have any number of constraints; each constraint has a *position* attribute that specifies the associated unification slot, and a number of values.

A clause therefore specifies the unification space, or number of slots, and gives constrained values to those slots. One or more *specification* nodes in each clause determine the correspondence between the XML input and unification slots in the template, and hence the application predicates.

4.2 Template Specifications

A template specification is well-formed XML, but unlike some template languages it follows a tree structure, rather than a sequence. Depth in the tree indicates subsequent operations and breadth allows the specification of alternatives. A *publish* attribute can appear anywhere in the tree, and its effect is to test that unification is complete, and if so mark that result as successful. For example:

```
<first>
<second tpl.publish="TRUE"/>
<third>
<fourth tpl.publish="TRUE"/>
</third></first>
```

This would find all instances of first…second and first…third…fourth that unified. (first, etc, are not of course valid node names)

There are three types of node in a template specification: Source Nodes, Navigation Nodes, and Matching Nodes. Source and Navigation Nodes may carry the attribute *tpl.mutable="TRUE"* in a *roundtrip* clause. This specifies that the node that can be removed or re-written when predicates from the client application are exported back into XML.

Source nodes simply name a node in the XML input tree. They cause the template to evaluate all nodes of that name from the current position in the XML input.

At present the language supports five types of navigation statement, two of which are concerned with performance management. Examples of the three core types are:

```
<tpl.select node="UML:ClassifierRole">

<tpl.selectFromChildren
          node="UML:AssociationEnd" position="0">

<tpl.moveUp steps="2">
```

The *tpl.select* node evaluates all nodes in the input tree with the specified node name, the example selects all *UML:ClassifierRole* nodes in an XMI tree.

The *tpl.selectFromChildren* node is intended to select child nodes from the present position in a specified order. Each occurrence of *tpl.selectFromChildren* specifies the position (i.e. index) and name of the child node to be selected. In this example the first occurrence of a UML:AssocationEnd node is selected.

The *tpl.moveUp* node simply moves the present position in the XML input tree up by a number of steps. This command has been included because it provides a very compact way of navigating certain tree structures, but there is a restriction on its use: it must never follow a mutable node. Nodes marked as mutable can unify with nodes that are not present in the XML input; allowing a step up from such a node may be non-deterministic, depending on how the mutable node has been reached, so this navigation is not permited from a mutable node.

There are three matching node types within the template language, and they each instruct the template processor to unify an element in the XML input tree with one of the unification slots, any previously specified constraints and, depending upon the process mode, a predicate retrieved from the client application. Examples of these are:

```
<tpl.match nodeType="ATTRIBUTE_NODE"
                    attribute="name" position="1">

<tpl.match nodeType="TEXT_NODE" position="0">

<tpl.match nodeType="MULTIPLE_ATTRIBUTE"
                    attribute="myLunch" tagIndex="1"
                    length="2" position="3" >
```

Each *tpl.match* node specifies the index of the unification slot that must be matched (*position*). The relationship between the unification slots and the client predicates is fixed, so this does not need to be specified. The node to be matched from the XML input is always the current node, reached by the last navigation. The first two match types unify the value of an attribute by name, or node text data, respectively. The third is more specialized and provides the ability to pack several parts of a predicate into a single XML attribute.

A multiple attribute match node unifies one value in an attribute list of separated values. For example, given the attribute *mylunch="fish,chips"* , the example above would correctly match the number of values in the attribute (length="2 ") and attempt to unify the value 'chips' (*tagIndex="1"*) with the template slot 3.

This is the core of the reversible template language. A small number of specialized language statements are omitted from this paper for reasons of space, they include the creation of xmi.ids and additional forms of constraint. Language features to support performance management and debugging are described below, but first the essentials of the language will be illustrated by some worked examples.

4.3 Examples

This section provides two examples of template clauses, which demonstrate how well the template language is able to hide the round-trip complexity. The first example is a complete *structure* clause:

```
<tpl.structure length="2">
   <tpl.constraint position="0">
       <tpl.value>data</tpl.value>
       <tpl.value>service</tpl.value>
   </tpl.constraint>
<tpl.specification>
   <tpl.select node="UML:Class">
   <tpl.match nodeType="ATTRIBUTE_NODE"
                        attribute="name" position="1">
   <UML:ModelElement.stereotype><UML:Stereotype>
   <tpl.match nodeType="ATTRIBUTE_NODE"
        attribute="name" position="0" publish="TRUE"/>
   </UML:Stereotype></UML:ModelElement.stereotype>
   </tpl.match></tpl.select>
</tpl.specification>
</tpl.structure>
```

There are two unification slots in the template, and these correspond directly to a client predicate with two values. The constraint section of this clause limits the first slot position to the values ‘data’ or ‘service’.

The specification searches all the nodes in the XML input for UML:Class nodes. For each node of this type it extracts the name attribute, which is unified with the second unification slot position. The template then searches child nodes for the stereotype (UML:ModelElelement.stereotype/UML:Stereotype) and it unifies the attribute name of the stereotype with first unification slot. Of course, this slot is constrained, so the only values that succeed are ‘data’ or ‘service’. The effect of this clause, therefore, is to search the XML input for UML:Class nodes with stereotype of ‘data’ or ‘service’ and, depending on mode, publish predicates of the form (data|service,name). The form of this template is very similar to other template languages, demonstrating that although reversible templates are theoretically quite different to conventional templates, their programming form can be made familiar.

The specification of mutable XMI nodes is essentially the same. The following is part of template clause for the Security Analyst Workbench:

```
<!—Slots:
   (tagname 1st_value className 2nd_value)(xmi.id)  -->
<!—Client use:
   (PermitAccess fromClass inClass toOperation)     -->

<tpl.roundtrip length="4" auxLength="1">
   <tpl.constraint position="0">
   <tpl.value>PermitAccess</tpl.value>
   </tpl.constraint>
   …
```

```
<tpl.specification>
   <XMI><XMI.content><UML:TaggedValue
                                 tpl.mutable="TRUE">
   <tpl.match nodeType="ATTRIBUTE_NODE"
                  attribute="tag" position="0">
     <tpl.match nodeType="MULTIPLE_ATTRIBUTE"
                  attribute="value"
                  tagIndex="0" length="2" position="1">
      <tpl.match nodeType="MULTIPLE_ATTRIBUTE"
                   attribute="value"
                   tagIndex="1" length="2" position="3" >
      <tpl.match nodeType="ATTRIBUTE_NODE"
                  attribute="modelElement" position="4" >
     <tpl.selectNode node="UML:Class">
      <tpl.match nodeType="ATTRIBUTE_NODE"
                   attribute="xmi.id" position="4">
      <tpl.match nodeType="ATTRIBUTE_NODE"
                   attribute="name" position="2"
                   publish="TRUE">
      …
```

The comments at the start of this extract describe the use of the unification slots and the resulting application predicate. This template matches an XMI tag, which is attached to a UML class. The name of the tag is 'PermitAccess' and the tag has two separated values (e.g. *PermitAccess="subject,object"*). The application predicate contains the same information as the tag, but also includes the name of the class in which the tag was declared (inClass). The first four template slots correspond to the values in the application predicate, and the fifth is used for the xmi.id of the class. The header to this clause specifies the number of unification slots, and constrains the first to the single value 'PermitAccess'.

The specification navigates directly from the document root (XMI) to a tagged value, which is marked as mutable. This specifies that any tagged values that match this clause will be re-written on export. This navigation identifies all possible tagged values, but only those that unify as far as the 'publish' tag at the end of this fragment will be rewritten.

The next three match statements unify the three elements of the tag (name plus two values) with their respective template slots. An important feature of this language is that the programmer is not concerned with the underlying operations. These statements are able to both extract data from the XMI tree and publish them to the client application, and also obtain predicates from the client and re-write it into an XMI tag, depending upon the operational mode of the template processor.

The fourth match operation unifies the modelElement attribute value with an auxiliary slot in the unification template (i.e. one that is not part of the application client's predicate). This value is the xmi.id of the class in which the tag is placed, and the next section of the template navigates to the corresponding class by selecting all the class nodes in the XML input, and matching the one with the correct xmi.id. The final match statement unifies the class name associated with this xmi.id with the third template slot. At this point the publish attribute tests if the unification process is complete, causing publication to the client, or the addition of a node to the XMI tree.

This fragment illustrates the extent that the underlying semantics of unification and reversible working are hidden from the template programmer, who is still able to think of the template as little more than a ‘select and publish’ script.

One notable feature of this fragment is the relative lack of constraint checking. In the Security Analyst Workbench, the two values in the tag are known types, the first corresponding to a class with a specific association to the class in which the tag appears, and the second to an operation within that class. It would be quite straightforward to navigate the XMI input tree and use the unification process to check that these values correspond to correct types. However, there are good reasons for avoiding these checks at this stage. Firstly, the template is specific to the tool that generated the XML input, but given that the template processor delivers tool-independent predicates, the type checking could be coded once, in the application, rather than separately for each supported tool. There is also a second consideration, which is that in its normal operation the template processor will often fail to unify, since it will attempt to match nodes and predicates that are not compatible. If constraint checking is included in the template, then badly constructed types will not unify, and will not be passed to the application. However, the result of a constraint failure in a template processor is silence, whereas constraint failures in the application can generate warnings to the user. The programming philosophy adopted has therefore been to specify the minimum in the template language, consistent with establishing an accurate relationship between the XML input and application predicates, and to carry out more extensive type checking in the application.

These examples illustrate the core language, two further issues, performance and debugging, add extra features, which are discussed in the next sections.

4.4 Performance Management

The main performance problem in template processing is the need to repeatedly scan all the nodes in a document. This problem occurs in the *roundtrip* example above. It is necessary to scan the entire document for UML:Class nodes, in order to match the xmi.id in the tag with the correct class name. Since Classes are in user-defined packages they can occur at any level of the XMI hierarchy, so it is not feasible to limit the search size by navigating from the tree root.

However, in UML templates, the types of node that are revisited often in this way are a relatively limited number of fixed design points, primarily the classes and objects. If it were possible to simply remember the location of these nodes then these auxiliary searches could be made much more efficient. This, quite simply, is what the performance management statements in XRound implement. There are two statements, one that records fixed points, and one that navigates to previously recorded nodes. For example, in the two examples above, the first, which identifies specific Classes, could include the statement:

```
<tpl.registerNode>
```

This registers the current node (in this case UML:Class), allowing it to be efficiently revisited later. The second example could then replace the selectNode navigation to a UML:Class with:

```
<tpl.selectRegisteredNode node="UML:Class">
```

The result is the same, but considerably faster. The only restriction on the use of these statements is that mutable nodes cannot be registered, and that nodes must be registered before they can be selected. Normal practice is to register nodes in early *structure* clauses, they can then be referenced anywhere in the template.

4.5 Debugging

Finally there are two important features in the language that are an aid to debugging – message and debug attributes – which can be added to any node.

The message attribute (*tpl.message="…")* can be included in any node, and sets a message for the template tree below that node. If any errors are issued from the processing of that section of the template, then the message will be included in the error report. It is good programming practice to include messages in every clause header; they provide useful comments and invaluable narrowing of the problem space when an error is reported.

The debug attribute (*tpl.debug="TRUE"),*is not intended to be a permanent feature of a template. Whenever a node is encountered with the debug attribute, the status of the unification slots is printed, together with the current template and XML nodes. Although this provides sufficient information to debug a template, usually the existence of debug output is sufficient: when a template fails the most common problem is detecting the node that caused the failure, so the most common use of this feature is as a probe to detect where a template succeeds or fails.

5 Limitations

There are few inherent limitations in the XRound language; the practical limits arise from two sources: variability in XMI between different UML tools, and limits to the scope of the current template processor.

Differences in XMI between UML tools was one of the main motivating factors in the design of XRound, and has been discussed at several points in the paper; different templates are required for different UML tools, but the use of a reversible template isolates the application logic from this variability. The XMI import behaviour of tools can also vary in detail; for example, some tools regenerate missing xmi.id fields, where others fail. The design of a template may therefore go beyond the need to understand the XMI tool dialect. Although this is an inconvenience, it has not yet proved a major problem, or required tool-specific language features.

At present the XMIsource processor is more limited than the language. Although it supports all the language features it does not support unification of nodes that are interdependent: for example it would not be able to unify both missing classes and associations between those classes. This is not an inherent limitation in the XRound language, but reflects the initial applications for XMISource, in which the class structure is stable but other elements of a model can be varied by analysis tools.

6 Conclusion

XRound adds a new dimension to template processing of models: the ability to transform data in both directions with a single descriptive template. Reversible template processing solves the problem of maintaining independence between UML and analytic tools, while retaining the benefit of easily scripted transformations. Reversible templates could provide a clean implementation mechanism for bi-directional transformations specified in QVT, and could help in the definition of model unification languages as well.

This paper has outlined the theory behind reversible templates, and presented a mature template language, XRound, that has been used in practice and is supported by a Java template processor. As well as including transformations the language includes performance management and debugging facilities.

The examples presented here illustrate the extent that the underlying semantics of unification and reversible transformation are hidden from the template programmer, who is still able to think of the template as a 'select and publish' script.

The successful implementation and use of a template processor demonstrates that it is feasible to implement a reversible processor that interprets the XRound template language, and the specification of the processor shows how straightforward the reversible interface is from the perspective of the client application.

References

1. Object Management Group. *Queries-Views-Transformations Specification*, available at http://www.omg.org, last accessed June 2005.
2. Object Management Group. *Model-Driven Architecture Specification,* available at http://www.omg.org, last accessed June 2005.
3. QVT-Merge Group, *Revised submission for MOF 2.0 Query/Views/Transformations RFP (ad/2002-04-10)*, 2004, www.omg.org.
4. ATLAS Transformation Language web page. http://www.sciences.univ-nantes.fr/lina/atl/, last accessed June 2005.
5. Xactium Inc. *XMF Reference Guide 0.1,* available at http://www.xactium.com, last accessed June 2005.
6. J. Herrington. *Code Generation in Action*, Manning, 2004.
7. TXL Web Page, available at http://www.txl.ca, last accessed June 2005.
8. M. Del Fabro, J. Bezevin, F. Jouault, E. Bertan, G. Guillaume. AMW: a Generic Model Weaver. In *Proc. IDM 2005*, July 2005.
9. L. Tratt. *The Converge Programming Language*, King's College Technical Report TR-05-01, 2005.

Towards General Purpose, High Level, Software Languages

Anneke Kleppe

Klasse Objecten, Netherlands
a.kleppe@klasse.nl

Abstract. A highly significant benefit of MDA is that it raises the level of abstraction at which the soft-ware developer is able to work. However, the languages available to the developer have not seen much change in the last decade. Modeling languages offer high level concepts, but the pre-dominant modeling language (UML) offers too little expressive power to be able to specify a system completely. Meanwhile, the level of abstraction of most programming language con-cepts is the same as 10 to 15 years ago. Although transformation tools may to some extent bridge the gap between modeling and programming languages, in practice the developer still needs to do both modeling and programming. This means switching between the two levels of abstractions, which is difficult for most people. We argue that a general purpose, high level, software language is necessary to get MDA adopted. This language will enable any developer to focus on the problem at hand while the supporting tools - transformation tools or generators- take care of the nitty gritty details. This paper introduces an early version of such a language, which brings together a number of powerful concepts from various sources: UML, OCL, design patterns, existing programming languages, and eventually aspect-oriented languages.

Keywords: Modeling language, programming language, UML, OCL, design patterns, domain specific languages, MDA, model transformations.

1 Introduction

MDA claims amongst others the following benefits: portability, interoperability, and productivity. These benefits are all very difficult to realise. In fact, almost every hype in the last two decades promised similar benefits, most of which were not or only to small extent realised. In our opinion the only real — but highly significant — benefit of MDA is that it raises the level of abstraction at which the software developer is able to work.

In the last decade, the expressive power of programming languages has developed slowly. The latest truly innovative concept that was incorporated in a programming language, is the interface, which dates from around 1994. On the other hand, there were some very interesting new developments, like the emerge of UML, design patterns, aspect-oriented languages, and last, but not least, OCL. Each of these developments offers new high level concepts: associations, patterns, aspects, collection iterators, etc. Few of these concepts have been incorporated into programming languages, which means that few of these concepts are easily available for the average software developer. If these concepts could be incorporated into a single language, this language would be very powerful, and would greatly add to the developer's ability to create the complex systems that customers demand.

This paper introduces an early version of Alan (short for A LANguage), which is a new software language that brings together a number of powerful concepts from various sources. Its aim is to bring more power to the software developer, and thereby realising one of the claimed benefits of MDA: increased productivity.

In [1] we defined 6 levels at which software development can take place. These levels are called Model Maturity Levels. Alan is a language that can be used to develop software at Modeling Maturity Level 4 or 5. At level 4 a model/program is a consistent and coherent set of texts and/or diagrams with a very specific and well-defined meaning. At level 5 the model/program contains enough information that the system can be generated completely. No adjustments need to be made to the resulting code.

Large parts of this paper, in particular sections 2 and 3, deal with the question why Alan was created. After we have explained the rationale behind Alan, we sketch the outlines of the language in section 4. Our plans for future work are presented in section 5. Section 6 contains some remarks on related work and some conclusions.

2 Rationale

We call Alan a high level, general purpose, software language. There are a large number of arguments for creating a this new type of language. We will encounter most of them as we explore the various parts of the term *high level, general purpose, software language*.

2.1 *Software* Language

Traditionally modeling and programming are viewed to be different. Differences like the ones in table 1 are commonly mentioned. Furthermore, traditionally there has been a gap between the analysis and design phase, and the implementation phase (the gap that two decades ago was supposed to be bridged by object orientation). Apparently, the expressive power of modeling languages stops somewhere along the line of the development process, and at that point the existing artefacts need to be transformed into one or more programming language artefacts, after which the development process can proceed.

Although we are living in a different era, many of the misconceptions of the previous age remain. It is for this reason that the question "what is the difference between a model and a program" pops up time and again. On the positive side, we see that the interest in MDA has brought us (at least some) agreement that both models and programs are descriptions of software systems. On the other hand MDA opens a wide chasm between platform specific and platform independent models, which at first glance appears to be just a different terminology for what used to be called models and programs. The problem here is the definition of the notion of *platform*.

Table 1. Perceived differences between modeling and programming languages

Modeling Language	Programming Language
imprecise	precise
not executable	executable
overview	detailed view
high level	low level
visual	textual
informal semantics	execution semantics
analysis by-product	end product

Fortunately, Atkinson and Kühne [2] have provided a definition of platform that crosses the divide. In their view a platform consists of the combination of a language, predefined types, predefined instances, and patterns, which are the additional concepts and rules that are needed to use the capabilities of the other three elements. Using this definition each model (or program) is bound to a certain platform. It is 100% platform specific to the language it is written in, and to the types, instances, and patterns associated with that language. In the same manner it is more or less independent of any other platform.

Anything written in this new type of language that we propose, is therefore 100% dependent upon the platform defined by such a language, and by its types and instances. If the language offers high level constructs, we may call it a modeling language. If the language is textual and/ or executable we might call it a programming language. Because this new type of language aims to combine the good aspects of both, we simply call it a *software development language*, or *software language*.

The question remains how to name the product written in a software language, should it be called *model* or *program*? The answer can be found in the fact that software languages build a bridge between programming and modeling. If a model is precise and executable, why not call it a program? Because the end result of software development has long been called *program* and because in the eyes of the developer the product written in a software language will be the end product, we choose to call it *program* as well.

2.2 *General Purpose* Language

Recently there has been much attention to the subject of domain specific languages [e.g. 3, 4]. In fact, some people argue that all MDA transformations transform domain specific languages to programming languages. In contrast, we think that there are sufficient grounds to introduce a general purpose language.

First, tool development for domain specific languages is at least as complex as for general purpose languages, whereas the potential number of users of these tools is much larger for general purpose languages. Thus, for economical reasons it is a good thing to have general purpose languages.

Second, domain specific languages are positioned as languages that can be developed by the domain experts themselves. If the supporting tools allow each expert to define his own domain specific language, the world would see a new version of the story of the Tower of Babel [5]. None of the experts, even in the same domain, would be able to understand the language built by one of his colleagues.

Third, the line between domain specific and general purpose, or as one might say *domain independent*, is as blurred as the line between platform specific and platform independent. For instance, there are arguments to say that graphical user interface design is a separate domain; only user interfaces contain buttons and windows. On the other hand, a graphical user interface is part of almost every software system, either in the form of traditional windows and subwindows, or in the form of webpages and frames or tables.

An excellent example of what can be called a domain specific language is the Enterprise Integration Patterns language by Hophe and Woolf [6]. This language is dedicated to the domain of asynchronous messaging architectures. Again, one might argue that with the current advent of web-based systems, asynchronous messaging is part of a large number of software systems. Should such a system be built using two domain specific languages, one for the user interface and one for the messaging, combined with an ordinary programming language for the rest of the system? We think not.

2.3 *High Level* Language

Frederick Brooks argues in his book the *Mythical Man Month* [7] that the productivity of a software developer is constant in terms of the number of statements he/she produces per time unit. He states: "Programming productivity may be increased by a much as five times when a suitable high-level language is used" (page 94 of the 1995 edition). In other words, the use of a high level language could bring us one of the claimed benefits of MDA: increased productivity.

Currently there are a large number of programming languages, all more or less on the same level of abstraction. When we compare them with the level of OCL expressions, it becomes clear that it is possible to increase productivity largely. Take for example the following OCL expression:

```
partners.deliveredServices->forAll(pointsEarned = 0)
```

This expression translates to the following Java code, which means that to implement one line of OCL seventeen lines of Java are needed, as well as an extra method.

```
Iterator it = collect5().iterator();
while ( it.hasNext() ) {
    Service i_Service = (Service) it.next();
    if ( !(i_Service.getPointsEarned() == 0) ) {
           return false;
    }
}
...
private List collect5() {
    List /*(Service)*/ result = new ArrayList( *Service*/);
    Iterator it = this.getPartners().iterator();
    while ( it.hasNext() ) {
           ProgramPartner i_ProgramPartner =
                            (ProgramPartner) it.next();
           result.addAll(
           i_ProgramPartner.getDeliveredServices());
    }
    return result;
}
```

Contrary to programming languages, modeling languages offer constructs at a high level of abstraction. The problem with today's modeling languages is that they do not have enough expressive power. For example, how can you create a system based on a UML model without a concrete syntax (called surface language in latest UML 2 specification [8]) for actions? You can not even indicate the creation of an object.

A combination of the high level constructs of modeling languages with the completeness of programming languages seems the obvious direction for future language developments. In this we feel supported by the words of Richard Soley, managing directory of the OMG, in his foreword to MDA Distilled [9]:

> "Somehow the high level abstraction allowed by programming languages does not always have significant run-time costs, so long as the precision of the abstraction allows complete definition of the algorithm."

High level abstraction is what we should aim for.

2.4 Additional Reasons

Next to the arguments that are packaged in the term *high level, general purpose, software language*, there are two additional reasons for the development of this new type of language.

First, an important aspect of MDA is that models should be transformed automatically. The artefacts of an earlier phase in the development process should no longer be transformed by hand into the format needed in the next phase, instead this part of the software development process is to be automated. There is a debate going on whether the developer should be able to manually alter the output model after the transformation. In our view manual manipulation is currently necessary for a number of reasons. However, when MDA technology has reached maturity, manual manipulation should be an exception, just as manual manipulation of compiler generated byte code or assembler code is an exception. Transformation tools are the compilers of the next decade.

A consequence of this is that either the language of the source model must be at least as powerful as the language of the target model, or the transformation engine combined with the transformation definition must add any lacking information. If the source language has insufficient expressive power, then the output model still needs to be manually developed further. In other words, either the languages used to develop software, or the tools, need to be brought to a higher level. Because it is always wise to investigate all options, it is best to do both.

A second observation is that a key development of the last decades: the emergence of design patterns, is not truly incorporated in either modeling or programming languages. Almost ten years ago (in 1996) Budinsky and others [10] wrote:

"Some developers have found it difficult to make the leap from the pattern description to a particular implementation, ... Others have no trouble translating the pattern into code, but they still find it a chore, especially when they have to do it repeatedly."

Since then not much has changed. At best the programming IDE offers some support, but the languages themselves have not changed. The support for patterns in the UML is not conveniently integrated. One has to draw a separate collaboration diagram to express that some classes play a part in a pattern. In practice, this is rarely done.

Hence, there are a large number of reasons to invest some effort in the development of this new type of software language. In the next section we will have a closer look at the requirements on such a language.

3 Requirements on General Purpose, High Level, Software Languages

Our new type of language should combine the positive aspects of both modeling and programming languages. So, what are the positive aspects that should be incorporated in software languages? To answer this question we take a second look at the characteristics in table 1. As shown in table 2, there are three negative aspects of modeling languages that should be avoided: non-executability of the model, the informal semantics of the modeling languages, and the model being a by-product. In the table these items have been crossed out. The other characteristics should be present in the new type of language.

Table 2. Characteristics of General Purpose, High Level, Software Languages

Modeling Language	Programming Language	Software Language
imprecise	precise	imprecise in early stages, precise in later stages
~~not executable~~	executable	executable
overview	detailed view	various levels of detail
high level	low level	high level
visual	textual	both visual and textual
~~informal semantics~~	execution semantics	execution semantics
~~analysis by-product~~	end product	end product

It is often considered convenient that a model/program may be imprecise in the early stages of development, but in later stages it should be precise. Furthermore, a model/program should be the end product and therefore, when it has reached the stage of precision, it should be executable. Thus a software language should have at least execution semantics. Different transformations may add different non-functional requirements to the end product. For instance, how the storage is arranged, whether logging is required, etc.

The visual syntax of modeling languages is often considered to be a positive aspect, but not all details can be shown visually in a convenient manner. Therefore it would be best to have a visual syntax and a textual alternative. The visual syntax will provide overviews, whereas the textual alternative may include many of the details of the program. In the same manner it is wise to provide two textual alternative syntaxes, one that is human readable, and one that is meant to be machine readable. Languages with multiple syntaxes have been created before, one example being the Mjølner BETA language [11] developed in the early nineties of last century.

Likewise, we want to keep several views, more and less detailed. This means not necessarily that the language should provide only two different views. A hierarchy of views, each level a bit more or less detailed than the following, is to be preferred. Traditional data flow modeling as described by Tom DeMarco and Edward Yourdon [12], has an excellent levelling mechanism that has been sadly missed in the UML, although in version 2 some leveling is possible. (Data flow models had a good way of zooming in.)

When listing these aspects it is sometimes difficult to determine whether a certain aspect should lead to the creation of language concepts, or whether it should lead to specific support in the development environment, the IDE. The following list is ordered by the influence the re-quirement has on on the language itself.

1. The modeling language should have a several concrete syntaxes that are all mapped onto the same abstract syntax.
2. The modeling language should have a clear semantics for precise programs. As long as the program is imprecise, the semantics may be unclear.
3. The program will be a complete functional description of the the system.
4. The developer should be allowed to be imprecise at certain stages of the development proces.
5. The program should be precise at most stages of development, specially during the last stages, when it is prepared to be used as input to a transformation.
6. If the program is imprecise it will very likely not be executable, but when all details are present, it should be executable (model simulator, model virtual machine)
7. The modeller should be able to have different views on the same program: overview and more detailed.

And surely, we should not forget the ultimate requirement, because otherwise the new language will be nothing more than a programming language in pictures: the language should provide constructs that are more abstract than current day programming languages.

4 ALAN: A Software Language

The goal of the Alan project is to gather and combine the concepts that are already well-known and have shown their use, into a format that is usable for a developer, not to create new software development concepts. Therefore, we use a number of different sources. The first source is UML [8, 13], for instance, the two-directional association is a powerful concept that is not present in current day programming languages. The second source is OCL [1, 14]. The possibilities OCL offers on collections are far more powerful than any programming language offers. The third source is design patterns [15], which beyond a doubt have been a landmark in software development. Currently, support for patterns can be found in some IDEs, but little support can be found in languages. The fourth source is found in current day programming languages, like Java, C#, and Python.

In the future we hope to include constructs from aspect-oriented languages as well. In our opinion a user of Alan should be able to define a number of cross-cutting aspects and weave them into a single output. Whether this effects the design of the language, or merely the design of the Alan IDE, is a question that remains to be answered.

What we present in this paper is an early version of Alan. Our ideas need to be developed further, but we feel it is already worthwhile to share them with the community and get some feedback. In the following sections we will present examples of language constructs from the above mentioned sources and explain how they are incorporated. The length of this paper does not allow us to be complete. Alan comprises more than just the examples given below. The language has been fully implemented and the Alan IDE and compiler are available as an Eclipse plug-in. This paper does not merely describe ideas, everything presented here was tested in our implementation, and shown to be feasible.

Alan's textual syntax is based on the Java syntax. One notable difference is that the equal sign is reserved for comparisons; assignments are denoted using the Pascal notation (":="). Alan's visual syntax is basically the same as the UML class diagram syntax. However, the semantics of Alan are much more strict than those defined for UML. The semantics of Alan are defined by a mapping to Java. This mapping is implemented as an MDA transformation.

4.1 UML Constructs in Alan

Apart from having used the syntax of the UML class diagram for the Alan visual syntax, we have borrowed a number of UML constructs. Some of which are explained in the following sections.

4.1.1 Associations

The UML association is a powerfull construct that needs to be implemented carefully. Specially, the two-directional association leads to complicated code, because setting the field that implements the association in the class at one end, must also ensure that the field that implements the other association end has the correct value.

In Alan, associations are always two-directional. A uni-directional association, as is present in the UML, is in Alan simply an attribute, or in ordinary programming terminology: a pointer. In our view, mixing the concepts of a reciprocal relationship and a reference, as is the case with the UML association, is confusing and, to some extent, overkill. As we explained elaboratedly in [16], if Eve is in the bag of Adam's girlfriends, then Adam must be in the of Eve's boyfriends, otherwise one could not speak of a relationship called friendship. If Eve insists in calling Adam her boyfriend, even though Adam does not regard Eve to be his girlfriend, then this fact can only be represented as a reference from Eve to Adam (or to a bag of not-interested boyfriends including Adam).

In Alan associations may have no more than two ends, and each association abides to the following characteristics, which we call the ABACUS rules.

- **Awareness:** Both objects are aware of the fact that the link exists.
- **Boolean existence:** If the objects agree to end the link, it is dissolved at both ends.
- **Agreement:** Both objects have to agree with the link.
- **Cascaded deletion:** If one of the objects is deleted, the link is dissolved as well.
- **USe of rolenames:** An object may refer to its partner using the role name provided with the association.

A simple example of associations depicted using the visual syntax can be found in figure 1. The Alan textual syntax that maps to the same abstract syntax is the following.

```
class Man {
   public Woman wife otherside husband;
   public Bag[Woman] girlfriends[1..10] otherside boyfriends;
   ...
}
class Woman
   public Man husband otherside wife;
   public Bag[Man] boyfriends otherside girlfriends;
   ...
}
```

The multiplicities in the figure need not always be part of the textual syntax for associations. In Alan the exact lower and upper bounds need only be present when they differ from 1..* or 1. The exact lower and upper bounds of multiplicities are considered to be invariants, which will be explained in section 4.2.3. Currently Alan does not support association classes. We are investigating if and how this could be done.

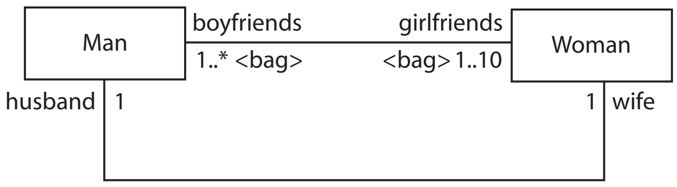


Fig. 1. An Alan association example

4.1.2 Enumeration Types

Typesafe enumeration types may not look like a powerful language construct, but in practice they come in very handy. However, implementing a typesafe enumeration type is not a simple matter. Joshua Bloch spends as much as 10 pages on this subject in his book Effective Java [17]. Still, when you have learned the trick, you see that every enumeration type can be handled in the same fashion. This is where the MDA transformation techniques can provide much assistance: a single line in a higher level language can be automatically transformed into a much more verbose text in a lower level language.

In Alan, typesafe enumeration types can again be written in the same simple way that they were written in C, while the translation of Alan into Java takes care of all the details of implementing the type safeness. An example of a declaration of a typesafe enumeration type:

```
enum myColor { red; white; blue }
```

This declaration may be part of a package, and thus have package scope, or it may be part of a class declaration, and thus have class scope.

4.1.3 Composite or Aggregate Objects

Another example of a higher level construct that is part of UML, but not part of any programming language, is the composite object. Much has been said on the semantics of the UML aggregate and composite, e.g. in [18]. The Alan composite object has deletion semantics; when the container is deleted, so are all its parts. Furthermore, although other objects may refer to the object by means of an attribute or association, this object may be part of no more than one composite object. Figure 2 contains a simple example of an Alan composite object. The code below is the textual alternative.

```
class Bike {
    public part Set[Wheel] wheels[2];
    public part Frame frame;
    ...
}
class Wheel {
    public part Tire tire;
    ...
}
```

As for associations, each part in a composite object is aware of the link to its owner. In fact, the role name *owner* may be used in the part object to indicate the containing object. If instances of the same class may be part of multiple composite objects, as for instance the class *Wheel* may be used as part in a class *Car* as well as the class *Bike*, the *owner* always refers to the one composite object that contains the specific instance.

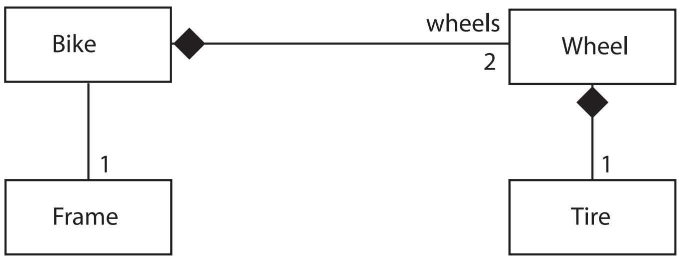


Fig. 2. An Alan composite object example

Although on the surface the notion of the composite object may appear to be nothing more than a more specialized version of the association, it serves a larger purpose. It enables us to easily specify the Visitor pattern, as explained in section 4.3.1. Furthermore, we are experimenting with a specific form of delegation, in which operations offered by the part objects become available in the composite object. A call to such an operation on the composite object will delegate the call to the part object or objects that implement it. For instance, when the operation *turn* is specified in the class *Wheel*, a call to a *Bike* object, as in *myBike.turn()*, will delegate this call automatically to both wheels.

4.2 OCL Constructs in Alan

The constructs that were defined in OCL, are incorporated in Alan completely, but Alan takes things one step further. For instance, OCL expressions may be used in statements, like assignments, and concepts like invariants are integrated in the textual syntax for the definition of a class. Again, this is not a new idea, it has been done in, for instance, Eiffel. What makes Alan different is the combination of existing ideas.

4.2.1 Primitive Types

The primitive types that are available in Alan are the same as the UML/OCL primitive types: *Integer, Real, String,* and *Boolean*. We believe that the abstraction level that Alan targets, has no need for low level details, like the differences between *char[]* and *String*, between *float* and *double*, and between *int* and *long*.

4.2.2 Collection Types and Iterators

OCL defines four collection types: *Set, OrderedSet, Sequence,* and *Bag*. These types are also available in Alan. Furthermore, the iterators defined on OCL collections, like *select, collect, exists,* and *isUnique*, are all part of Alan as well. In the future, we hope to augment Alan with a syntax for defining new iterators, to enable users to specify their own iterators in terms of existing ones. The example in section 2.3 shows clearly the power that OCL collection types and iterators bring to Alan. Because the available collection types reside at a much higher level of abstraction, Alan does not support arrays.

Furthermore, we are investigating two additional collection types: the *SortedSet* and the *SortedBag*. Both should sort their elements based on the "natural" order of the element's type, defined by the *equals, greater-than,* and *smaller-than* operations.

4.2.3 Invariants and Pre- and Postconditions

As can be expected, Alan supports OCL invariants and pre- and postconditions completely. In the textual syntax they are integrated in the class definition, as in the following example.

```
class Man {
   public Woman wife = oclUndefined otherside husband;
   public Bag[Woman] girlfriends[1..10] otherside boyfriends;
   public Integer age = 0;
   private Real moneyEarned = 0;
   inv ofAge: age < 16 implies wife = oclUndefined;
```

```
    public void work()
    pre: age >= 14
    post: moneyEarned =
                moneyEarned@pre + 100 and notassert(girlfriends-
                mult)
    {
         ...
    }

}
```

All invariants are checked at postcondition time of all operations of the class, as well as after the setting of the value of an attribute. If the developer needs to speed up processing, he can choose to check only some invariants or none at all by using the keyword *notassert*. The *notassert* takes as parameters the names of the invariants that should not be checked. If no parameters are given, then none of the invariants will be checked.

In section 4.1.1 we mentioned that the upper and lower bound of the multiplicities for an association are considered to be invariants. This means that the bounds will also be checked at postcondition time of any operation execution. When this check is not necessary, this too can be indicated in the *notassert* clause. As name of the invariant the role name of the association end concatenated with "-mult" may be used, as in *girlfriends-mult*.

4.2.4 Derivation rules

OCL derivation rules can be expressed in Alan as well. Before the execution of an operation of the class, the value of the derived attributes is determined, during execution this value remains the same. The next example contains two examples of derived attributes: *frontwheel*, and *speed*.

```
class Bike {
   public part Sequence[Wheel] wheels;
   public derived Wheel frontwheel := wheels->first();
   public derived Real speed :=
         frontwheelsize * frontwheel.revolutionsPerSec;
     ...
}
```

4.3 Design Patterns in Alan

Currently, only the most popular patterns are incorporated in Alan. We expect to add to this list in the future.

4.3.1 Visitor

The visitor pattern in Alan is linked to the composite object concept. Only composite objects can be visited. In general composite objects have the form of a directed graph; only some of the graphs will have the form of a tree. The graph is not necessarily acyclic, therefore the implementation algorithm ensures that the execution will terminate. Figure 3 shows an example, where node 3 is about to be visited twice, once caused by the link with node 1 and once caused by the link with node 5. Here the algorithm ensures that node 3 will not be visited because of its link with node 5.

The textual syntax for the visitor pattern is shown in the next example. The visual syntax shows only the fact that the class *BikeVisitor* is a visitor to *Bike* objects, the details are not shown visually.

```
     class BikeVisitor visits Bike <breadthfirst> {
         String visit(Bike bike) {
                String brand;
                before {
                      brand := bike.getDefaultBrand();
                }
                after {
                      brand := resultOf(bike.frame);
                      brand + resultOf(bike.frontwheel);
                      brand + resultOf(bike.wheels->last());
                      return beautify(brand);
               }
     }
     String visit(Wheel wheel) {
            after {
                 return wheel.brand + resultOf(wheel.tire);
            }
     }
     String visit(Frame frame) {
            return 'frame';
     }
     String visit(Tire tire) {
           return 'dunlop';
     }
     String beautify(String brand) {
           ...
     }
 }
```

The keyword **`visits`** indicates that instances of this class are visitors of composite objects of type *Bike*. Directed graphs can be transversed in several ways. The most

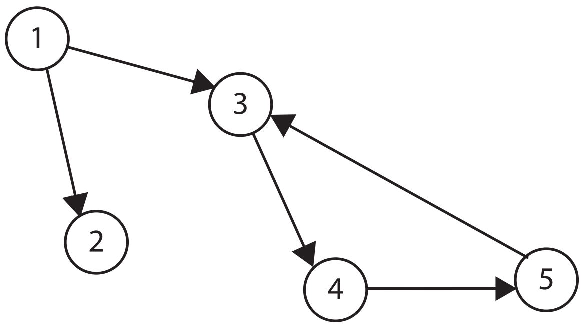


Fig. 3. An Alan composite object as directed graph

important traversal methods are breadthfirst and depthfirst. The visitor in the example visits *Bike* objects breadth-first, that is it visits all direct parts of the *Bike* instance before visiting the parts of the *Wheel* instances. Alan also supports visitors that use the depthfirst method.

Any visit operation is divided into two parts. Before visiting parts of an composite instance the statements in the *before* clause are executed. After visiting parts of the instance the statements in the *after* clause are executed. Classes that are part of the composite, but are not composite objects themselves, are the leaves of the tree or directed graph. For them the distinction between the before and after clause cannot be made.

Visit operations may have a result. In the example all visit operations have a String result. This result can be used in the after clause of the visit operation that visits the containing object, using the keyword **`resultOf`**. The type of the value that is returned by **`resultOf`**, is the type of the corresponding visit operation. If, in the example, the visit of a Tire instance would have re-turned an Integer value, then the expression **`resultOf`**`(wheel.tire)` in the visit operation for Wheels would have returned an Integer as well (which would have resulted in a type error).

The visitor class need not define a visit operation for all nodes in the composite graph. When a visit operation for a certain type of node is not present, the traversal algorithm simply proceeds. Next to the visit operations, visitors may have ‘normal’ operations as well. The operation *beautify* is an example.

Visiting may start at any node within the directed graph. You simply create a visitor instance and tell it to start visiting a certain object. If the object is not within the composite object that the visitor was defined for, a type error occurs. The next example shows how the *BikeVisitor* can be used.

```
myBike := ...;
visitor := new BikeVisitor( );
if (visitor.visit(myBike).equals( “someString ”)) { ... }
System.out.println(visitor.visit(myBike.frontwheel));
visitor.visit(myBike);
```

4.3.2 Singleton

Another popular design pattern is the singleton pattern. This pattern is easy to use in Alan, one extra keyword suffices, as shown in the next example. The output of this example is, of course, twice the String ‘this is the unique instance of MyFirstSingleton’, followed by two occurrences of 'changed name of unique singleton'. Note that the singleton user is unaware of the fact that it is using a singleton instance, which is different from the use of a singleton in e.g. Java, where you cannot use "new", but you have to use a specific class method to get the instance.

```
singleton class MyFirstSingleton {
    public String name :=
       'this is the unique instance of MyFirstSingleton';
    ...
}
```

```
class SingletonUser {
    public useSingleton() {

            MyFirstSingleton a := new MyFirstSingleton();
            System.out.println(a.name);
            MyFirstSingleton b := new MyFirstSingleton();
            System.out.println(b.name);
            a.name := 'changed name of unique singleton';
            System.out.println(a.name);
            System.out.println(b.name);
    }
}
```

4.3.3 Observer

The third pattern that is incorporated in Alan, is the Observer pattern. The key to this pattern are two predefined operations that are available on every class: *observe* and *disobserve*[1]. The *observe* operation takes as parameters the object to be observed, and the name of the operation that should be called whenever a change occurs in the subject. This operation must be defined in the class of the observer, and it must have one parameter, which type is the type of the object to be observed. The *disobserve* operation takes as parameter the object that should no longer be observed. The next example defines a simple observer that observes two other instances, one of type *Subject*, and one of type *OtherSubject*.

```
class MyFirstObserver {

    public start() { Subject mySubject1 :=
            new Subject(); OtherSubject
            mySubject2 := new OtherSubject();

            System.out.println('>>>observing
            mySubject1');self.observe(mySubje
            ct1, 'uponChange1');
            mySubject1.attr := 'blue';
            mySubject1.attr := 'red';

            System.out.println('>>>observing
            mySubject2');self.observe(mySubje
            ct2,
            'uponChange2');mySubject2.attr :=
            "black';mySubject1.attr :=
            'green';

            System.out.println('>>>DISobserving
            mySubject1');self.disobserve(mySubje
```

1 The name *disobserve* is still under debate. Other options considered are *unlink*, and *unsubscribe.* The term *disobserve* is closely related to the term *observe*, which seems preferable.

```
                ct1);mySubject1.attr :=
                'white';mySubject2.attr := 'yellow';
        }
        public uponChange1(
        Subject mySubject ) {System.out.println("The value of
        Subject.attr is "
                                        + mySubject.attr);
        }
        public uponChange2(OtherSubject mySubject)
                {System.out.println("The value of
                OtherSubject.attr is " + mySubject.attr);
                }
                ...

    }
```

The output of operation *start* is:

```
>>>observing mySubject1
The value of Subject.attr is blue
The value of Subject.attr is red
>>>observing mySubject2
The value of OtherSubject.attr is black
The value of Subject.attr is green
>>>DISobserving mySubject1
The value of OtherSubject.attr is yellow
```

4.4 Programming Language Constructs in Alan

Most of the constructs known in programming languages are also present in Alan, although some have been discarded because their level of abstraction was considered too low. A few programming language constructs in Alan deserve more attention. They are explained in the following sections.

4.4.1 Generic Types

Only few programming languages support generic types. Alan offers full support, in fact, the collection types are considered to be predefined generic types. Generic types may be defined independently of any other types, but, as the next example shows, one may also define a new generic type by inheriting from one of the collection types.

```
class MySetType [ TYPEVAR ] extends Set [ TYPEVAR ] {

    public attr : TYPEVAR;
    public setAttr : Set[TYPEVAR];

    ...

    public oper1(newV : TYPEVAR) : TYPEVAR {
           attr := newV;
           return res;
    }
}
```

4.4.2 Visibility and Set and Get Operations

In Alan, explicit definition of *get* and *set* operations for attributes is not necessary. When these operations are not defined for a certain attribute, they will be generated according to the visibility of that attribute. If the developer wants to execute some extra statements and/or checks in the *get* or *set* operation, he may define the operations himself. This is similar to properties in C#.

4.4.3 Loops

In Alan, the OCL iterators are available for many of the cases where you would normally use a loop construct in a program.Therefore, the need for loop constructs in Alan will be much less than in one of today's programming languages. However, we still need a loop construct for some special cases, like simply doing the same thing for a fixed number of times.

Alan provides two primitive loop constructs: the for-loop and the while-loop. The for-loop must be used with two Integer values separated by two dots, that indicate the lower and upper bound of the number of times the body of the loop must be executed. The while-loop takes a boolean expression as guard, as in the next examples.

```
for(1 .. someInt) { ... }
while(someBoolean) { ... }
```

5 Future Work

As explained earlier, this paper describes an early version of Alan. Many aspects of the language still need to be fleshed out. We have already mentioned the inclusion of constructs from aspect-oriented languages, the support for association classes, and support for other patterns. Another issue are the libraries that should accompany this language, which should include extra predefined types like the *SortedSet* and *SortedBag*. Key to the success of Java has been the enormous number of predefined types available. We are convinced that languages like Alan will need a similar set of libraries, which should also be full of higher level constructs ready to use.

The Alan IDE is currently being implemented as an Eclipse plug-in. Because of the ongoing work the Alan IDE is not yet available for a large audience, but if you want

Table 3. The realisation of the requirements by Alan

Software Language	Alan
imprecise in early stages, precise in later stages	yes, visual syntax allows imprecision, textual syntax does not
executable	yes, by translation to Java code
various levels of detail	not yet established
high level	yes!
both visual and textual	yes!
execution semantics	yes!
end product	not yet, but going strong

to have some idea of the Java code being generated, you can take a look at the Java code generated by the OCL tool Octopus [19]. The code for the associations and for the OCL expressions is the same.

What remains is a check to see whether the requirements we defined in section 3 are met by Alan. Alan meets almost all of the requirements, as shown by table 3. We strongly believe that it is only a matter of time (and hard work) to realise the remaining requirements, and that in the near future Alan will be the general purpose, high level software language that we envisioned.

6 Conclusion and Related Work

On the topic of related work we can be short. A lot of work is being done in the area of domain specific languages, e.g. [3, 4], including work in the area of Executable UML [20], as well as in the area of formal specifications [21, 22], but virtually none is done in the area of general purpose, high level software languages. One might argue that we too have created a DSL, one dedicated to programming, but in our view this argument stretches the concept of domain far too much. If programming itself can be identified as a domain, then COBOL, Ada, and all other programming languages should also be called DSLs.

Currently, it is possible to create a complete visual representation of Java, or any other programming language in UML. This fact does not in any way diminish the need for a general purpose, high level software language. The essence of Alan is not that it combines a visual and a textual representation, as stated in section 3, this has been done successfully before. Instead Alan's merits lie in the fact that it incorporates higher level concepts, and makes them available to the programmer in a way he or she is likely to understand.

Please note that the creation of general purpose, high level, software languages will not make informal models obsolete, it will just raise the level of abstraction. This is a normal phenomenon in the history of any technology (or culture). Old ways become the stepping stone for future developments. In the same way current day middleware will, in time, take its place on one of the lower levels of our technology stack.

Raising the level of abstraction does not mean that the old ways are crooked or misformed. One has to build a wall by putting in the first stone, which will support all the other stones. Therefore, the first few stones need to be solid and well fitted. By creating Alan we do not criticize any other technologies, for instance, Java 5 has done some good work on enumeration types. We simply argue that it is time to start building the next layer of stones, and that the next layer should include general purpose, high level software languages.

We hope that Alan will not be the only software language at this level. Software development needs the boost that this new type of language can give. Therefore, in the future we hope to see a large family of general purpose, high level, software languages.

References

[1] Anneke Kleppe, Jos Warmer, *The Object Constraint Language Second Edition, Getting Your Models Ready for MDA*, 2003, Addison-Wesley
[2] C. Atkinson and T. Kühne, "A Generalized Notion of Platforms for Model Driven Development", in *Model-driven Software Development - Volume II of Research and Practice in Software Engineering*, ed. S. Beydeda and V. Gruhn, Springer Verlag. 2005.
[3] Jack Greenfield and Keith Short with Steve Cook and Stuart Kent, *Software Factories, Assembling Applica-tions with Patterns, Models, Frameworks, and Tools*, Wiley, 2004
[4] Alexander Felfernig e.a., *UML as Domain Specific Language for the Construction of Knowledge-Based Con-figuration Systems*, InInternational Journal of Software Engineering and Knowledge Engineering, Vol.10 No. 4 (2000) pp. 449 - 469, World Scientific Publishing Company
[5] The Bible, Genesis 11: 1-8
[6] G. Hophe and B. Woolf, *Enterprise Integration Patterns*, Addison-Wesley, 2003.
[7] Frederick P. Brooks, The Mythical Man-Month, Addison-Wesley, 1995
[8] *UML 2.0 Superstructure Specification*, OMG document ptc/04-10-02, October 2004
[9] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise, *MDA Distilled, Principles of Model_Driven Ar-chitecture*, Addison-Wesley, 2004
[10] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu, *Automatic code generation from design patterns*, IBM Systems Journal, 35(2), 1996.
[11] *Object-oriented environments: The Mjolner approach*, Editors: Jorgen Lindskov Knudsen (Aarhus University, Denmark), Mats Lofgren (Telia Research AB, Sweden) , Ole Lehrmann Madsen (Aarhus University, Denmark), Boris Magnusson (Lund University, Sweden) , Prentice Hall, 1994
[12] T. DeMarco, P.J. Plauger, *Structured Analysis and System Specification*, Prentice Hall, 1985
[13] *Unified Modeling Language (UML) Specification: Infrastructure*, OMG document ptc/04-10-14, October 2004
[14] UML 2.0 OCL Specification, OMG document ptc/03-10-14, October 2003
[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns, Elements of Reusable Ob-ject-Oriented Software*, Addison-Wesley, 1995
[16] Anneke Kleppe and Jos Warmer, *Wed Yourself to UML with the Power of Associations, part 1 and 2*, online publication at http://www.devx.com/enterprise/Article/28528 and http://www.devx.com/enterprise/Article/ 28576
[17] Joshua Bloch, *Effective Java, Programming Language Guide*, Addison-Wesley, 2001
[18] Brian Henderson-Sellers and Franck Barbier, *Black and White Diamonds*, in "UML" '99 - The Unified Modeling Language: Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 1999. Proceedings, Editors: R. France and B. Rumpe, LNCS 1723, pp. 550 - 565, Springer-Verlag, 1999
[19] Octopus: OCL Tool for Precise Uml Specifications, available from http://www.klasse.nl/ english/research/octopus-intro.html
[20] Stephen J. Mellor and Marc J. Balcer, *Executable UML, A foundation for Model-Driven Architecture*, Addison-Wesley, 2002
[21] A. Nymeyer and J-P. Katoen, *Code generation based on formal bottom-up rewrite systems theory and heuris-tic search*, Acta Informatica, Vol. 8, pages: 597 - 635, 1997
[22] V.M. Jones. Realization of CCR in C. in Bolognesi T, Van de Lagemaat J and Vissers C.A. (eds.), LOTO-Sphere: Software Development with LOTOS, pp. 348-368, Kluwer Academic Publishers, 1995.

Toward Standardised Model to Text Transformations

Jon Oldevik, Tor Neple, Roy Grønmo, Jan Aagedal, and Arne-J. Berre

SINTEF Information and Communication Technology,
Forskningsveien 1, 0373 OSLO, Norway
{jon.oldevik, tor.neple, roy.gronmo, jan.aagedal,
arne.berre}@sintef.no
http://www.sintef.no

Abstract. The objective of this work is to assess the qualities of the MOFScript language, which has recently been submitted to the OMG as a proposed model to text transformation language. This is done by identifying requirements for this type of language and evaluating the MOFScript language with regard to these. The language is presented along with a tool implementation and compared with the alternative languages submitted to the OMG Model to Text RFP.

1 Introduction

Ever since dawn of software modelling, technologies have been around to provide mappings from software models to useful technology platforms, such as databases, implementation languages etc. Along with the maturity of the modelling domain, the standardisation of modelling languages and technologies such as UML and the Meta Object Facility (MOF)[1], and the adoption of these technologies in practical use, the need for standardising transformation and mappings of these models has become apparent.

This need is currently being addressed through the ongoing standardisation activities in OMG concerning model to model transformations (MOF Query, View and Transformations – QVT)[2] and model to text transformations[4]).

The MOFScript language has been submitted as a proposal for a model to text transformation language to the OMG. This paper identifies different requirements for model to text transformation languages and evaluates the MOFScript language and tool against those requirements. There are three competing languages to MOFScript which are also discussed with regard to the requirements. The rest of this paper is structured as follows: Chapter 0 gives some background in the area of model to text transformation. Chapter 0 describes a set of requirements for model to text transformation languages. Chapter 0 describes the details of the MOFScript language and tool and gives a brief evaluation. Chapter 0 describes related work and chapter 0 concludes.

2 Background

Traditionally models have been used in software development to define and understand the problem domain or the different aspects of a system's architecture. After the modelling, one dove into implementation of the system without updating the models based on the actual implementation that was made. This issue is remedied within the MDA paradigm where the models are the prime artefact. From these artefacts large portions of the source code for the system can be generated.

The issue of generating code from models can be abstracted to the term *model to text* transformations as opposed to *model to model* transformations. The goal is to be able to create textual artefacts based on model information. Textual artefacts include other things than source code, such as various types of documentation.

Typically a code generator will not be able to generate all of the code that is needed to implement a system. Certain facets of a system, e.g. the static parts, are well suited for code generation, while there are challenges in modelling the more dynamic parts, for instance method bodies of classes. This means that parts of the source code for a system will be generated from the models while other parts will be hand crafted. The ability to protect the hand crafted parts of the code from subsequent code generation passes is important. In some cases one even may want to update the model based on changes made in the source code. One may say that the same issue is relevant for document generation, where one typically may want to add writings that are not part of the model to the resulting document.

The task of writing model to text transformation definitions will probably not be carried out by all software developers. However, it is important that the language used for such definitions is as easy to use as possible, e.g. by sharing properties with common programming or scripting languages. The usability of the language can made better through good tool support including features such as code completion, a feature present in most integrated development environments (IDEs) for normal programming languages today.

3 Requirements for Model to Text Transformation Languages

In the Request for Proposal (RFP) for MOF Model to Text Transformation[4], a set of requirements to such languages is identified. These are high-level requirements that provide a framework for defining a language that will fit the OMG way of thinking and align well with already adopted OMG specifications. The essential requirements that must be met include the basic ability of generating text from models, specifying transformations based on metamodels, the ability to specify complex transformations, the ability to allow multiple input models/metamodels for transformations, support for text manipulation functions, and reuse of existing standards, such as QVT, Object Constraint Language (OCL)[12], and MOF.

In addition, there are other obvious requirements, such as the ability to generate text to files, and the ability to query model element properties, which need to be supported by a model to text transformation language.

We acknowledge the OMG requirements as essential basic properties, and extend with a set of additional requirements that we deem desirable for a model to text transformation language. Some of these were previously identified in [6].

1. *Structuring:* The language should support structuring and control of text generation. This means that it should be possible to specify structures that orchestrate a set of finer grained text generations.
2. *Control mechanisms*: It should provide basic control flow mechanisms. This implies that it must be possible to provide the semantic equivalent of loops and conditional statements.
3. *Mix code and clear text*: It should provide a simple way of combining transformation code (logic), model data, and clear text. It shall also provide a way of converting model data to strings and use this in produced text
4. *System services*: It should provide support for string manipulation functions. It should also provide the ability to interact with system services or library functions, e.g. inquiring about the current date and time.
5. *Ease-of-use*: The concrete language should show similarity with existing well known approaches in order to be easy to use (such as programming or scripting languages). Adhering to aspects of the forthcoming QVT standard concrete syntax may also be beneficial.
6. *Expressiveness*: Finally, it should provide expressiveness to support expected domain needs; sufficient expressiveness may be a trade off with respect to ease of use.

The above described requirements are related to qualities of the transformation language. Some pertinent aspects for model to text transformation need to be addressed outside the scope of the language itself, and rather in the architecture of the tools implementing the language. Specifically, this is valid for change management scenarios such as incremental generation, reverse engineering, and round-trip engineering. Support for traceability between model elements and generated text can facilitate these aspects. Traceability links are also independent of the transformation language itself, although the language may open for defining configuration properties that control the nature of traceability links. The language itself may also define mechanisms to control the processing of such links.

The following chapter will look at the MOFScript language in detail and discuss how it meets the identified requirements.

4 The MOFScript Language

The MOFScript language has been defined to answer the needs of a standardized model to text transformation language, as called for by the OMG in the MOF Model to Text Transformation RFP[4]. MOFScript is based on the QVT-Merge[3] specification in terms of metamodel extensions and lexical syntax.

A MOFScript rule is a specialisation of QVT-Merge operational mappings, and MOFScript constructions are specialisations of QVT-Merge constructions. The main goals with the language are to provide ease-of-use, minimize additions to QVT, as well as providing flexible mechanisms for generating text output. It is presumed that a source metamodel is defined on which one can perform queries. This is analogous to QVT, while the explicit definition of a target metamodel is not required in MOF-

Script. MOFScript can be classified as an imperatively oriented language with traditional scope rules and with optionally typed variables.

The following sections look at the details of the MOFScript language, a tool that implements it, and an evaluation considering the requirements identified.

4.1 The Lexical Language

Module: MOFScript transformations are packages within modules, which defines the properties and rules of a transformation. A module is denoted with the keyword "textmodule" followed by a name for the module. The initial part of the module is identical to a QVT mapping rule module except for the keyword which is called *textmodule* as opposed to the QVT *module*.

```
textmodule UML2WSDL (in uml:uml2)
```

A module can import and reuse rules defined in other modules. This is achieved with the 'access library' statement.

```
access library Uml2wsdl ("uml2wsdl-lib.m2t")
```

Rules: The transformation rules are defined with a name and a potential context type. A rule may have a return type. It may also have a guard. The syntax is similar to that of QVT mappings. There is no specific keyword associated with the declaration of rules.

```
uml.Class::classToJava () {
  // statements    }
```

The guard for a rule is defined in the same manner as guards in QVT, using a 'when' clause.

```
uml.Class::classToJava ()
  when {self.getStereotype() = 'Entity'}
{   // statements   }
```

Files and Output Printing: Files are the most important kind of output device for text. A file is declared with a set of properties: name, extension, directory, and type. The File name property must always be present. File name and directory can be specified as separate properties. The directory portion may also be embedded in the file name property.

A file can be used implicitly or explicitly in output statements. For example, if a file device is declared, subsequent output statements will use that device as the target. If several file devices are declared, the latest one is used by default. If a specific device is the target, it can be referenced by its name. Output printing is done by using standard print functions or escaped output. The standard print functions are either 'print' or 'println', which output an expression (the latter adds a new line character (for the appropriate platform / encoding).

A couple of other utility print functions are defined, to provide easier whitespace management: newline (or nl), tab, or space, followed by an optional count integer. Standard String escape characters (\n\t) are also legal within String literals.

```
file ("an-output-file.txt")
<%
  This text is generated to the output file.
%>
```

```
file f2 ("AnotherFile.txt")
file ("Yet-another.txt")
println (" Now, I am writing to the file 'yet-another.txt'")
f2.println (" Now, I am writing to the file 'AnotherFile.txt'");
```

Escaped Output: Escaped output provides a different and in some cases simpler way of providing output to a device. Escaped output works similar to many scripting languages, e.g. Java script.

Escaped output is signalled by escape characters, beginning and ending of an escape. Basically, it is a print statement that can subsume multiple lines and be combined with all expressions that evaluate to a string. Escaped text is signalled by the characters '<%' to start an escape and '%>' to end an escape. Note that whitespace is copied to the output device.

```
uml.Operation::bindingOperation () {
  <%
   <operation name="%> self.name <%">
    <soap:operation soapAction="%> nameSpaceBase + self.name <%"
style="document"/>
     <input>
            <soap:body%>
             if (self.ownedParameter.size() > 0) {
              <% parts="%> self.getParameterOrder() <%"%>
                  }
                  <% use="literal"/>
         </input>
         <output>
          <soap:body%>
           if (self.returnResult.size() > 0){
                <% parts="response"%>
           }
            <% use="literal"/>
         </output>
   </operation>
  %>
}
```

Properties: Properties are used in the same manner as in QVT. They can be defined at the module level or within a rule. There are two types of properties; local properties which are constants within a module or a rule, and configuration properties which are global properties that may be used in many transformations.

```
property javaPackageName = "org.sintef"
```

A property cannot be modified after its declaration. It is typically used in output statements.

```
<% The Java package name is: %> javaPackageName <% Nothing more,
nothing less %>
```

Variables: Variables are defined and used as in QVT. They can be defined globally for a module, or locally within a rule. A variable can have an assigned value when declared, which can be modified during its lifetime.

```
var exportCounter = 0
var modifiableName:String = "temporary name";
var storedNames:Hashtable;
```

A variable can be typed. The standard OCL types are used (String, Boolean, Integer, Real). In addition, the Collection types List and Hashtable are introduced in MOFScript, which are similar to List and Hashtable classes in Java. These are used for holding sets of values during transformation execution, e.g. to temporarily store pre processed information that is needed several times during generation.

```
var packageNames:List
var packageIdList:Hashtable
self.ownedMember->forEach(p:uml.Package) {
  packageNames.add (p.name)
  packageIdList.put (p.id, p.name)
}
if (packageIdList.size () > 0) {
 <% Listing the package names that does not start with ‘S’ %>
 packageIdList->forEach (s:String | not(s.startsWith(“S”)) {
   <% Package: %> s
 }
}
```

Iterators: Iterators in MOFScript are used primarily for iterating collections of model elements from a source model. A for-each block expression defines an iterator expression which also has a block of executable expressions.

It works similarly to forAll in OCL or the shorthand iterator expressions from QVT.

```
-- applies to all objects in the collection
-- of type DBTable that has a name that starts with ‘a’
c.elements->forEach(e:DBTable | e.name.startsWith(“a”)) {
  -- statements
}
```

If-Then-Else: If-expressions provide basic functionality for controlling execution based on logical expressions. An if-expression has a condition and a block of statements that are executed if the condition is met. It might have a set of else conditional branches and an empty else branch. It basically has the same semantics as any conventional programming language if statement.

```
uml.Package::interfacePackage () {
 if (self.name = "Interface Model") {
  self.ownedMember->forEach(p:uml.Package) {
     p.interfacePackages()
  }
 } else {
  stdout.println (“Error in model.”)
 }
}
```

Invoking Rules: Text transformation rules are invoked either directly or as part of expressions.

```
uml.Package::interfacePackages () {
 if (self.getStereotype() = “Service”){
  file (rootdir + self.name.toLower() + ".wsdl")
  self.wsdlHeader()
  self.ownedMember->forEach(i:uml.Interface) {
     i.wsdlPortType()
  }
  self.wsdlFooter()
 }
}
```

Return Results: Text transformation rules may have return results. This is most useful for defining *helper* functions.

```
uml.TypedElement::getType () : String {
 if (self.type.name.equalsIgnoreCase("string"))
  result = "xsd:string"
 else
  result = self.type.name
}
```

Library Functions: MOFScript defines a set of functions to support manipulation of strings and collections. The string manipulation functionality is similar to that provided in Java. In addition it defines utility functions to manage white space, and functions to retrieve system date and time. It currently does not provide additional functions to interact with the system environment.

4.2 The MOFScript Tool

This section gives an overview of the MOFScript tool, a tool supporting the definition and execution of model to text transformations using the MOFScript language, implemented as an Eclipse plug-in, which is available for download[8].

The architecture: The MOFScript tool is developed as two main logical architectural parts: tool components and service components (see Fig. 1). The tool components are end user tools that provide the editing capabilities and interaction with the services. The services provide capabilities for parsing, checking, and executing the transformation language. The language is represented by a model (the MOFScript model), an Eclipse Modeling Framework (EMF) model populated by the parser. This model is the basis for semantic checking and execution. The MOFScript tool is implemented as an Eclipse plug-in using the EMF plug-in for handling of models and metamodels.

The Service Components consist of these component parts: The *Model Manager* is an EMF-based component which handles management of MOFScript models. The *Parser and Lexer* are responsible for parsing textual definitions of MOFScript transformations, and populating a MOFScript model using the Model Manager. The parser is based on antlr[7]. The *Semantic Checker* provides functionality for checking a transformation's correctness with respect to validity of the rules called, references to metamodel elements, etc. The *Execution Engine* handles the execution of a transformation. It interprets a model and produces an output text, typically to a set of output files. The *Text Synchroniser* handles the traceability between generated text and the original model, aiming to be able to synchronize the text in response to model changes and vice versa.

The Tool Components consist of these component parts: *The Lexical Editor* provides the means of editing transformations, invoking the parser, checker, and the execution engine. The *Result Manager* is responsible for managing the result of a transformation in a sensible way, such as integrating result code files in an Eclipse project.

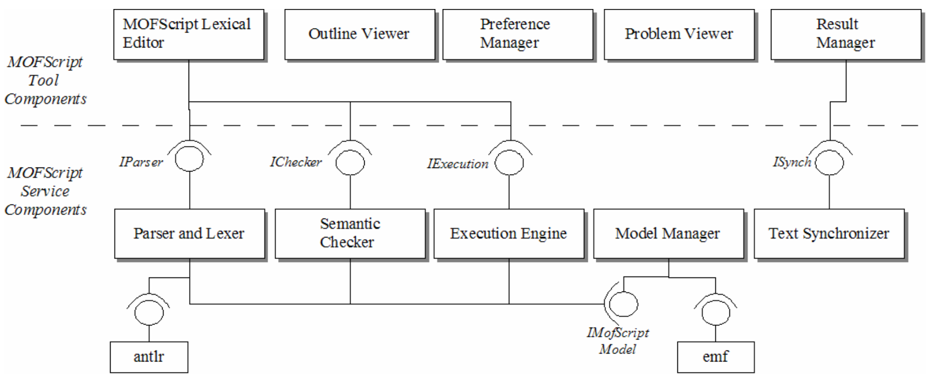


Fig. 1. MOFScript component and tool architecture

The *Outline Viewer*, Preference Manager, and *Problem Viewer* provide simple graphical components to guide the user in writing and executing transformations.

The Model: This section shows the model design used in MOFScript. It is used to generate the EMF model representation of MOFScript, which in turn is utilized by the parser (which produces instances of it) and the Execution Engine. Fig. 2 shows the main MOFScript model structure.

- The *MTTTransformation* class represents a MOFScript transformation module. It has a name, it imports a set of other transformations, it may have parameters, and variable/constant declarations. Finally, it has a set of transformation rules.
- A *TransformationRule* represents a rule (or a function) within a MOFScript transformation. A rule owns a set of statements (*it is a MTTStatementOwner)*, and may have parameters and a return type. Rules define the behaviour of a transformation.
- The *MTTImport* class represents the import of external transformations for a transformation module (MTTTransformation). It is represented by a name and a URI.
- The *VariableDeclaration* class represents variable or constants (properties) for a module (or for statement owners. It has a name, a type, a constant flag and a calculatedValue property (to store the value of simple variables). Basic OCL variable types are supported (String, Boolean, Real, Integer), as well as List and Hashtable types.

A transformation rule consists of different kinds of statements that define the operational logic of the rule:

- The *PrintStatement* class represents printing to a file or to standard output. Print statements produce output towards either the current output device or an explicit prefixed output device. A special syntactic kind of print statement is escape statements, which provide direct output without a print/println command. A print / println statement without prefix will produce output to the current output device. The same will an escaped output statement do.
- The *ResultAssignment* represents the assignment of a value to the result of a rule.

- The *IteratorStatement* represents a loop that iterates on a collection of elements (typically a collection of model elements in a source model). For each element in the collection, a set (a block) of statements is executed.
- The *IfStatement* represents a normal if statement with a condition. It may have an else branch.
- The *GeneralAssignment* represents an assignment of a value to a variable
- The *FunctionCallStatement* represents an explicit call to a rule.
- The *FileStatement* represents the declaration of an output file context, which can be used to print output with print statements.

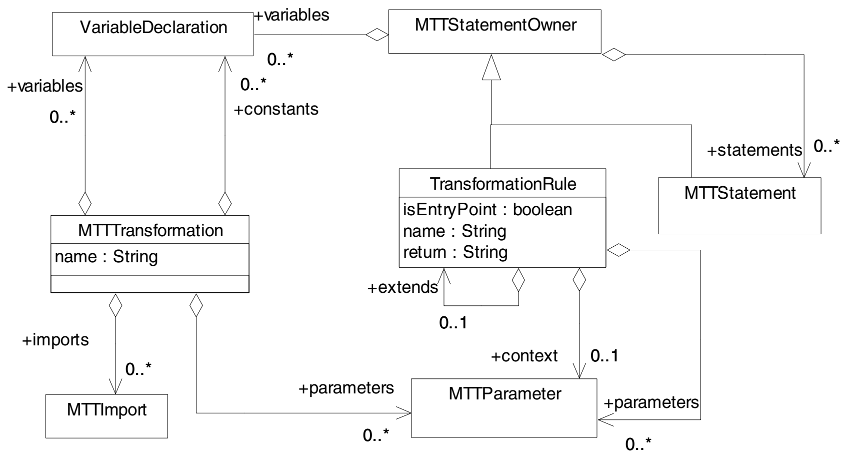


Fig. 2. MOFScript model structure

The User Interface: The MOFScript tool UI is provided through Eclipse editor functionality. It encompasses, as depicted in Fig. 1, a lexical editor, an outline viewer, a configuration manager, and a problem viewer. The lexical editor provides syntax high-lighting and useful code completion associated with the currently active meta-models.

4.3 Change Management

Change management is a highly pertinent issue for model to text generation and involves several aspects, such as management of manual changes to generated code, management of changes to source models, handling reverse engineering and model synchronization, tracing mode information to generated code, and round-trip engineering.

MOFScript does not specify any language-specific mechanisms to support traceability, but a metamodel has been defined to potentially manage the traces from a source model to target files.

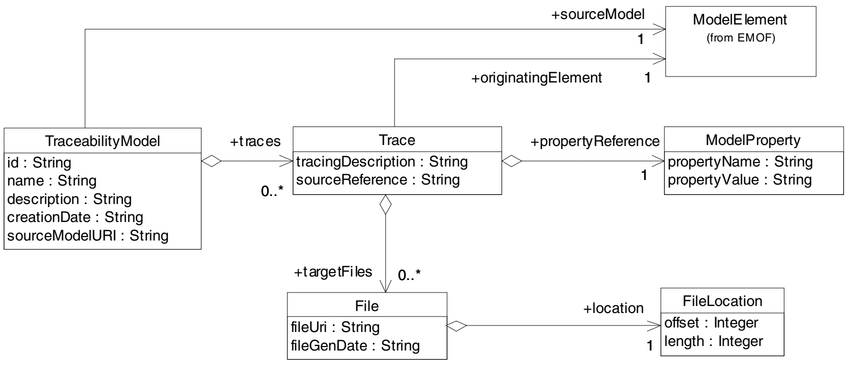


Fig. 3. Model to Text Traceability Model

Traces in this model are managed per model element of the source model. For each (relevant) model element, links are managed to files and file locations within those files that reference the model element.

Source model changes: Changes to source models are only an issue if the already generated text/code has been manually modified, and not yet synchronized with the model. In this case, traceability information can be used to synchronize modified text with newly generated.

Traceability of model information in generated code: In order to support manual changes in generated files, a kind of traceability mechanism that associates generated text with model elements must be in place.

A commonly used solution for handling this is to provide *tags* in the generated code, which establishes the relationship with a part of the text (such as a property or a method) with a model element. This kind of scheme will define a set of relationships between the generated text and the model. These kinds of tags are however dependant of the target language, so they cannot be standardized. The MOFScript language must offer a flexible, user-defined tag mechanism, which can be used as delimiters in the generation (typically, these are embedded as part of comments, Javadoc or similar).

Another solution is to manage traceability information in a separate model, referencing the source model and the generated text files.

4.4 Evaluation of MOFScript

MOFScript supports the basic requirements described in the OMG RFP, i.e. it is capable of generating text based on MOF M2 metamodel specifications, it supports manipulation of strings, it can generate files, etc.

This section looks at the additional requirements and assesses how MOFScript meets those requirements.

1. *Structuring*: Structuring is supported through definition, composition, and invocation of transformation rules. A transformation can import other transformations, and a rule can invoke other rules in a structured manner.

2. *Control mechanisms*: Control mechanisms are provided by supporting iteration over model collections as well as for conditional processing (if-statements).
3. *Mix code and clear text*: Code, clear text output, model references and other expressions can easily be combined in print and escaped print statements.
4. *System services*: MOFScript provides the ability to interact with a limited set of system services, based on what is considered most useful. This is open for future extensions.
5. *Ease-of-use*: The MOFScript language was originally designed to have a look and feel similar to existing programming languages. It then migrated toward the look and feel of the QVT textual concrete syntax in order to establish the compatibility at that level.
6. *Expressiveness*: The expressiveness of MOFScript is kept on an as-simple-as-possible level and defined on a need-to-have basis. Its resulting concrete syntax is therefore quite simple, yet expressive enough to handle complex model to text transformation tasks.

The current MOFScript tool[8] realisation implements all aspects of the language described here, except for guards on transformation rules and change management functionality.

5 Related Work

The most relevant work in this context is the alternative languages submitted to the OMG MOF Model to Text RFP process in three other proposals.

Basically, all the proposals meet the general requirements of the RFP. This chapter will describe the proposals and discuss their positions concerning the additional requirements identified in this paper.

5.1 MOF2Text Partners and the Template Language Specification (TPL)

The MOF2Text partners consist of Mentor Graphics, Pathfinder Solutions, and Compuware Corporation. Their submission [9] presents an imperative approach which also focuses on aspect-oriented concepts. The concrete language is called Template Language Specification (TPL). TPL defines *Patterns*, which are basic structuring mechanisms, similar to modules. They can extend and import other patterns or block libraries. It defines *Methods* as invokable units, which are evaluated in the context of an active output buffer. Methods have parameters, and seem to be defined without any particular metamodel context. A special kind of parameter is a *Literal Parameter*, which allows for sending complex literal expressions as parameters (these may have parameters themselves). File statements declare active output buffers. The language provides basic control statements in terms of ‘if’-statements and ‘for’-statements, and variable assignment in terms of ‘let’-statements.

This submission focuses heavily on the use of aspects as a central mechanism. These are defined within the metamodel, but are not so visible in the concrete syntax.

The listing below gives a brief overview of how this submission meets the identified additional requirements.

1. *Structuring*: Structuring is supported in terms of Patterns and Methods. In addition, the concept flexible literal parameter, allows for complex parameters to be passed to methods.
2. *Control mechanisms*: Control mechanisms are provided in terms of for-statements and if-statements.
3. *Mix code and clear text*: Clear text can be combined with model expressions to produce output.
4. *System services:* The MOF2Text submission defines a set of operations for string and buffer manipulation. It also defines the notation of context operations, which support the notion of functions on well-known objects (e.g. introspection). Additional environment operations are not mentioned.
5. *Ease-of-use*: TPL uses a tagged-based syntax, with square brackets that defines keyword tags in an XML-like manner: `CREATE SCHEMA [SCHEMANAME(schema)/]`; The MOF2Text metamodel is aligned with MOF and OCL, but does not seem to consider the QVT metamodel. The concrete syntax (TPL) is not aligned with QVT. It appears to be less than easy-to-use.
6. *Expressiveness:* The TPL concrete language provides a tag-based syntax, providing advanced template-like substitution mechanism with the literal parameters. There is however discrepancies between the concrete syntax and the metamodel described, e.g. in lack of aspect support. The language seems to provide a high degree of expressive power.

5.2 Interactive Objects (IO)

The IO submission [10] presents a declarative approach with a two-phase transformation strategy. The first phase is the calculation of a target text model based on transformation rules. The second phase is the serialization of text from this model. The submission defines a range of special structuring concepts: Artifact, Section, and Slot define the things to generate. In practice, artefacts represent files. Sections represent (method-like) parts of those artifacts. Slots are properties of an artifact, which are assigned at runtime. An artifact defines parameters (typically with types from the source metamodel) similar to an operation. A Section defines a kind of method which is used by an artifact. It always returns a sequence of its respective type. Pool parameters represent references to collections of objects (typically from the source model), assigned using an OCL expression. The concept Record is used to define functions that cannot produce output text. These are used to group construction of multiple artifacts. The concept Transformation defines an entry point for a transformation, relates to the source metamodels, and invokes artifact sections. The actual text production is performed by templates linked with artifacts and sections. These are defined externally (in separate template files) and provide output text combined with usage of section and artifact slots (properties).

The listing below gives a brief overview of how this submission meets the identified requirements.

1. *Structuring*: The IO language defines a particular kind of language to structure text transformations, controlled by artifacts, sections and slots. These effectively represent model elements of a text/file model, which in turn is used to generate text using text templates.
2. *Control mechanisms*: Control mechanisms are implicit in matching mechanisms of templates. No explicit mechanisms seem to be defined.
3. *Mix code and clear text*: Code and text output are combined within template files.
4. *System services*: The IO language does not specify any ability to interact with system services.
5. *Ease-of-use*: The IO language seems designed to match the artifact metaphor used in IO‘s tool (ArcStyler). This gives it a distinct structure and style to match the IO graphical transformation structure. It does not seem to reuse any part of the QVT metamodel or syntax. It does, however, reuse OCL for expressions. The language architecture may cater for a high learning curve, but may also be easy to use when first learned.
6. *Expressiveness*: The IO language proposes a declaratively tuned language, where artifact structure is defined independently of template files. Although defining a lot of specialised concepts, it the approach seems flexible and providing for sufficient capabilities.

5.3 Tata Consultancy Services (TCS)

The TCS submission [11] defines an imperative approach based on templates rules. Template rules can be structured into modules. A template rule consists of output text (clear text) in combination with control logic (such as for loops) and metamodel references. Other template rules are invoked explicitly. Template rules may have guards and may override other template rules. A module can extend other modules. Conceptually, the TCS submission seems similar to MOFScript, although different in look and feel.

The listing below gives a brief overview of how this submission meets the identified requirements.

1. *Structuring*: Structuring is provided in terms of modules that can import other modules, and rules that can invoke other rules.
2. *Control mechanism:* Control mechanisms are provided in terms of for-loops and guards on rules.
3. *Mix code and clear text:* Output combines clear text with model reference expressions.
4. *System services:* The TCS language specifies a library for string manipulation and setting current output file. It does not provide other means of system library interaction.
5. *Ease-of-use:* The TCS language defines a quite simple syntax that combines template code with clear text output, similar to a scripting language. The TCS language reuses MOF and OCL concepts, but does not seem to relate to QVT. It appears to be as an easy-to-use language.
6. *Expressiveness:* The TCS language is based on simple principles of templates providing output and calling other templates. It seems to have necessary expressiveness to support complex text transformations.

5.4 Summary

Based on this comparison we can learn that it is not that easy to differentiate the concepts in the different submissions. Although different in the flavour of concrete language, the conceptual differences are not that big. Clearly, concepts such as the aspect-oriented focus of the MOF2Text proposal are clearly distinct. The two-phase transformation focus of the IO proposal appears conceptually distinct in its more declarative approach, as well as the separation between structure and output templates. The TCS proposal is conceptually very close to what is proposed in MOFScript with most distinctions at the concrete syntactical level.

6 Summary and Conclusion

This paper has described the MOFScript language and tool with evaluation against a set of criteria that we see as important for a model to text transformation language standard. These criteria are used also to evaluate the other proposals for the MOF OMG Model to Text transformation RFP.

The MOFScript language and tool allow a user to define model to text transformations from instances of arbitrary metamodels. The language has party based on the definitions from the current QVTMerge specifications, thus keeping the family of transformation languages as similar as possible.

The implementation of the MOFScript tool as an Eclipse plug-in allows for its usage as part of a MDD workbench that can include modelling tools, model to model transformations and model to text transformations in addition to the standard programming environment. We believe that it is necessary to have the model to text transformation tool (at least the execution part) as a tightly integrated part of the MDD tool chain or workbench. Otherwise the number of tool changes needed to complete a full MDD iteration will become too large, causing developers to loose focus through the context changes. This need should of course be balanced with the important issue of choosing the best tool for the task.

Currently the MOFScript tool and language are being used in pilot projects within the MODELWARE[*] project in order to assess the ideas and to provide feedback and input to the further development. Early feedback indicates that some of the QVT like syntax is somewhat unfamiliar to the developers.

From the current status of OMG submissions, it is not easy to see exactly which direction the standard for model to text transformation is headed. A standard needs to accommodate several requirements, but most importantly, it needs to be *usable* and *used*. Time will show if the involved parties are capable of arriving of a best-of-breed integration that will be able to meet this requirement.

Acknowledgment. The work presented here has been carried out within the MODELWARE project (IST Project 511731)[*].

[*] MODELWARE is a project co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2002-2006). Information included in this document reflects only the author's views. The European Community is not liable for any use that may be made of the information contained herein.

References

1. Meta Object Facility 2.0, MOF 2.0 Core Final Adopted Specification, OMG document ptc/03-10-04
2. MOF 2.0 Query / Views / Transformations RFP, OMG document ad/2002-04-10
3. QVT-Merge Group, Revised submission for MOF 2.0 Query/Views/Transformations RFP version 2.0, OMG document id ad/2005-03-02, http://www.omg.org/cgi-bin/apps/doc?ad/05-03-02.pdf
4. MOF Model to Text Transformation Language RFP, OMG document ad/04-04-07, http://www.omg.org/cgi-bin/apps/doc?ad/04-04-07.pdf
5. MOFScript Revised Submission to the MOF Model to Text Transformation RFP , OMG document ad/05-05-04, http://www.omg.org/cgi-bin/apps/doc?ad/05-05-04.pdf
6. J. Oldevik, T. Neple, "*Model Abstraction versus Model to Text Transformation*", position paper at European Workshop on MDA (EWMDA)
7. ANTLR, ANother Tool for Language Recognition, http://www.antlr.org/
8. MOFScript Eclipse plug-in, http://www.modelbased.net/mofscript
9. MOF2Text Partners Revised Submission for MOF Model to Text Transformation Language RFP, OMG document ad/2005-05-14, http://www.omg.org/cgi-bin/apps/doc?ad/05-05-14.pdf
10. Interactive Objects MOF Model-To-Text Transformation Language RFP – First Revised Submission, OMG document
11. Tata Consultancy Services, Submission for MOF Model to Text Transformation Language, OMG document ad/2005-05-15, http://www.omg.org/cgi-bin/apps/doc?ad/05-05-15.pdf
12. UML 2.0 OCL Specification (OCL 2.0), OMG Document ptc/03-10-14

On Relationships Between Query Models

Dominik Stein, Stefan Hanenberg, and Rainer Unland

University of Duisburg-Essen, Essen, Germany
{dstein, shanenbe, unlandR}@cs.uni-essen.de

Abstract. Queries on software artifacts play an important role in novel software development approaches, such as Aspect-Oriented Software Development and OMG's Model Driven Architecture. Keeping them separate from the modifications operating on them has proven to be beneficial with respect to their comprehensibility and their reusability. In this paper we describe what relationships can exist between such stand-alone queries. These relationships allow the combination of existing queries to form new ones, enabling developers to come up with abstractions for common selection patterns.

1 Introduction

Queries are an essential software artifact in modern software development techniques, such as Aspect-Oriented Software Development (AOSD) [12] and OMG's Model-Driven Architecture (MDA) [18]. Experiences gained in AOSD have shown that dealing with queries as first-class entities (i.e., as autonomous entities that can exist without further reference to any other entities) helps to understand and reason about the purpose and the effects of aspect-oriented adaptations, and improves the reusability of aspect-oriented code [9] [8]. From these experiences we anticipate that the same benefits will be brought forth to MDA if model queries are handled as autonomous artifacts. However, to actually achieve these benefits, developers require appropriate means to relate two (or more) queries to each other, so that to combine two queries to form a new one (for example).

In this paper, we present a set of binary relationships that can be established between two query models. With help of these relationships, developers can compose new queries from existing ones. Furthermore, the relationships help developers to reason on the semantic dependencies between queries; thus, providing developers with the means to identify recurring "selection patterns" that they may abstract from for future use.

The remainder of this paper is structured as follows: First of all (section 2), we give a brief overview of «Join Point Designation Diagrams» [23], the model query notation used throughout this paper, and introduce the running example for the motivation section. After that, we discuss why and what relationships between query models are needed and beneficial (section 3). Then we give a general description of the relationships that may exist between query models, and detail their semantics, giving a supplementary example (section 4 and 5). After elucidating related work in section 6, we conclude the paper with a summary and a short discussion in section 7.

2 Overview to JPDDs and to the Motivating Example

«Join Point Designation Diagrams» (JPDDs) have been introduced in [23] and [22] as a notation to specify query models. The notation is based on Unified Modeling Language (UML) [20] user-model symbols, and provides means to specify lexical constraints on element names (in terms of name and signature patterns) as well as on the structural context and/or the behavioral context those elements reside in (e.g. the (non-)containment of features in classes and of classes in packages; or the (non-)existence of paths between classes (or objects) in the class (or object) hierarchy and between messages in the call graph, etc.). Fig. 1 gives a graphical overview to the most important symbols, which may be arranged in analogy to standard UML symbols as stated in the UML specification [20].

Apart from selection constraints, JPDDs may specify those elements that are to be exposed for further adaptations (e.g. for model transformations). These elements are given identifiers, which are prepended by a question mark (?) and entangled in angle

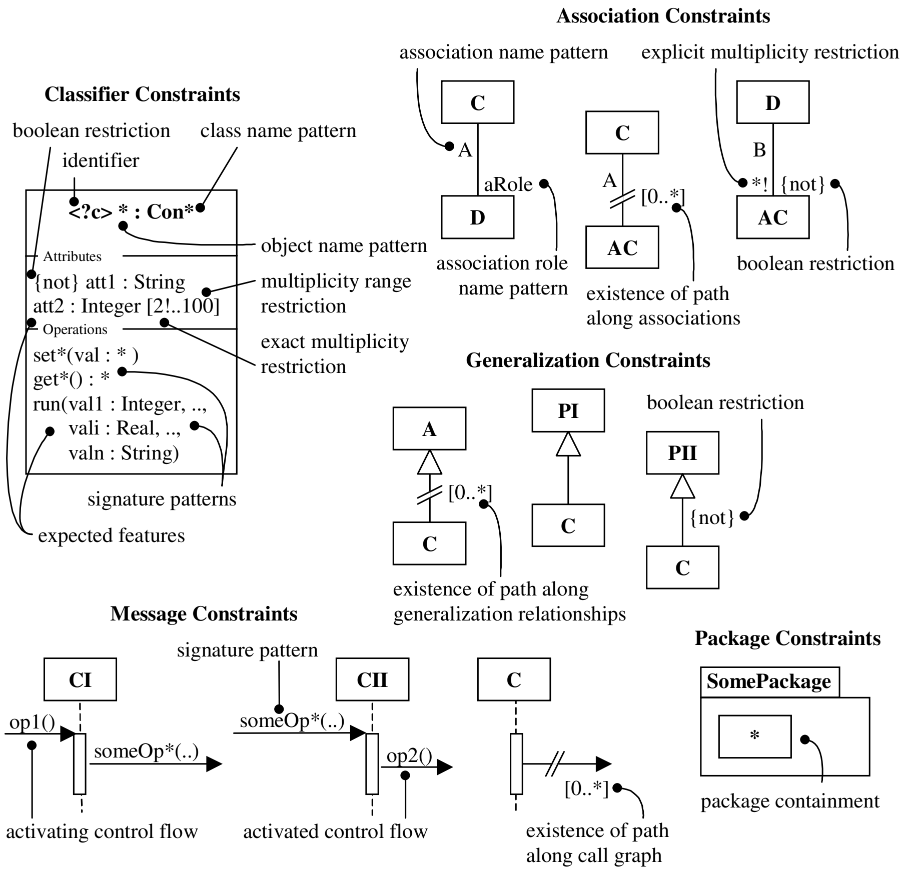


Fig. 1. Specifying selection constraints with «Join Point Designation Diagrams» (cf. [23])

brackets (< >), and which are listed in a parameter box in the lower right corner of the JPDD.

In the following section, multiple sample JPDDs will be given and explained in further detail (see Fig. 2, Fig. 3, Fig. 5, and Fig. 6). These JPDDs are used to represent a pointcut as it is used to implement the decorator design pattern [7] with AspectJ [14]. The example is adopted from [15]; its goal is to monitor the progress of reading some input stream. To do so, it hooks onto method `read` of any `InputStream` object, and ties the monitoring dialog to a GUI component – in this case, a `JComponent` object from the `javax.swing` package.

3 Motivation

Relationships between query models are needed for the following reasons: First of all, they are necessary to concatenate selection constraints that are rendered by different diagrams and/or different notations. In Fig. 2, for example, the object diagram on the left outlines the structural selection constraints of a query (select all objects `<?is>` of (sub)type `InputStream`), while the interaction diagram on the right renders the behavioral selection constraints (select all method invocations `<?jp>` to operations named "read" (of objects being (sub)type of `InputStream`), taking an arbitrary number of parameters `(..)`, and returning one parameter of type `int`). Usage of relationships in this way permits the representation of each selection constraint in its most appropriate manner. For example, control flow-based constraints can be represented in terms of interaction diagrams; state-based constraints can be represented using state chart notation; and data flow-based constraints can be represented with help of activity diagrams; etc. In the context of MDA, these relationships allow each part of a query to be specified in terms of the diagram and/or notation which it is eventually going to be applied to.

Apart from that, relationships enable developers to come up with abstractions over recurring selection patterns, thus facilitating their comprehension and allowing their reuse in different settings. For example consider Fig. 3, which visualizes an application of the prominent "wormhole selection pattern" [14] as it is regularly used in AOSD. The wormhole selection pattern in AOSD is used to perform adaptations on the behavior of programs which make use of context information from an earlier step in the execution process. The wormhole shown in Fig. 3, for example, selects all

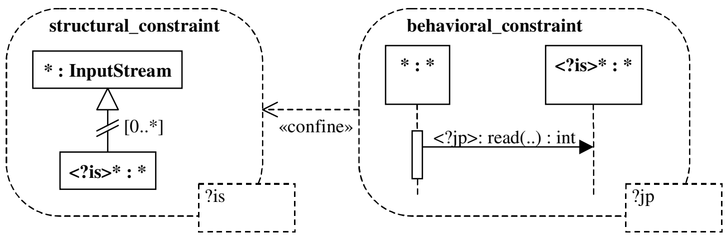


Fig. 2. Concatenating selection constraints of different nature (e.g. structural and behavioral selection constraints, in this case)

method invocations (`<?jp>*(..) : *`) that occur in the control flow (⫽▸) of any (other) method (`*(..) : *`) invoked on an object of (sub)type `JComponent` (as defined in package `javax.swing`); the receiver object of the latter is exposed by the query model for further adaptations, or for use by another query model. Factoring out the wormhole selection semantics into a separate query (and giving it an appropriate name) helps developers to recognize the intention as well as the effects of the query more easily[1]. Furthermore, the same wormhole can now be reused by multiple (other) queries.

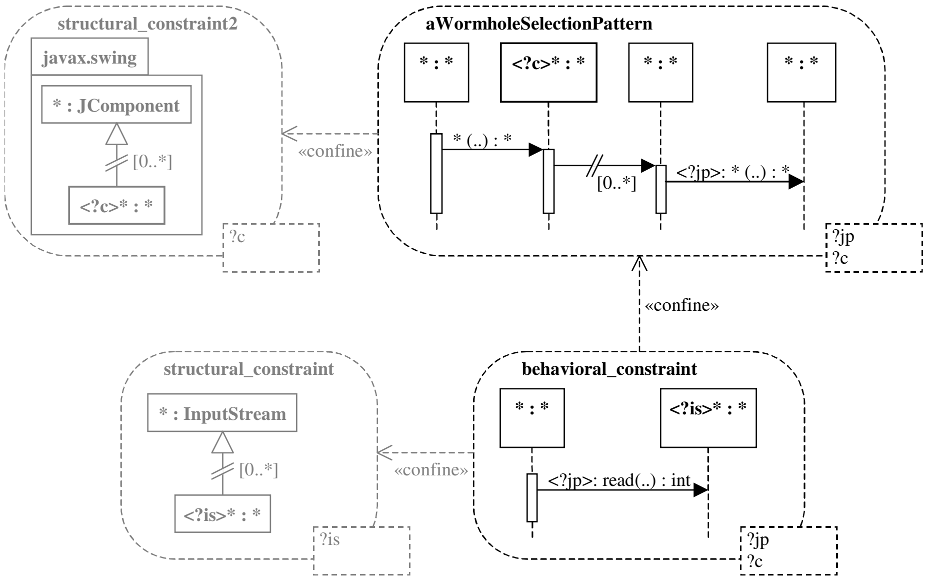


Fig. 3. Grouping selection constraints according to intention and effects, and finding abstractions for common selection patterns to facilitate comprehension and enhance reusability

We expect to see many selection patterns like that to arise in the MDA context. Over time, developers are going to recognize that they are using similar selection patterns over and over again. They are going to find suitable idioms for those selection patterns, and we anticipate them to ask for suitable means to abstract over those patterns and to reuse them in most different contexts. The wormhole pattern described above is one example of such recurring selection patterns. It is used to reason about the entities that have participated in (e.g. initiate) a certain task, and to select them for future adaptation (i.e. transformation). Another interesting example is concerned with the reasoning on and selection of recurring method calls; this will be considered in the subsequent chapter 4 in subsection 4.5.

[1] It may be advisable to abbreviate the (explicit) relationships to the structural selection constraints using the (implicit) short hand described in section 4.3 (see Fig. 5), in this case. That way, we have to deal with just two rather than four JPDDs.

4 Relationships

Having elucidated necessity and benefits of query model relationships, we now take a closer look at the nature of these relationships and define them more rigorously. We introduce three kinds of relationship: «union», «confinement», and «exclusion». These relationships combine the selection criteria of one query model with the selection criteria of another query model; however, every time in a different manner.

In the following, we refer to the including query model (e.g. JPDD_D in Fig. 4) as the "client" query model, and to the included query model (e.g. JPDD_A, JPDD_B, or JPDD_C in Fig. 4) as the "supplier" query model (in compliance to common UML parlance).

4.1 Mapping Rules

All combination relationships can be annotated with mapping rules indicating what identifiers from the including (client) query model are matched to what identifiers from the included (supplier) query model for unification purposes. Mapping rules are enclosed in curly braces (`{ }`) and are prepended by a lower-case Rho "ρ" in the following manner "`ρ{clientid=supplierid}`" (see Fig. 4 for an example). In case no explicit mapping rules are given, correspondence is assumed for all identifiers of same name. If explicit mapping rules are given, though, any identifier from included (supplier) query model and/or the including (client) query model *not* being mentioned in the mapping rules is considered irrelevant for the union, confinement, or exclusion, respectively.

Mappings may only be established between identifiers referring to the same type of element (i.e. classes, objects, attributes, operations, parameters, stimuli, etc.). Furthermore, mapping rules may only be specified for identifiers of the included (supplier) query model if they are exposed in the query model's parameter box. Any other identifier specified inside the included query model is not accessible from outside the query model. In contrast to this, identifiers specified inside the including (client) query model may be involved in mapping rules. This is because the inclusion is assumed to happen in the namespace of the including query model (the including

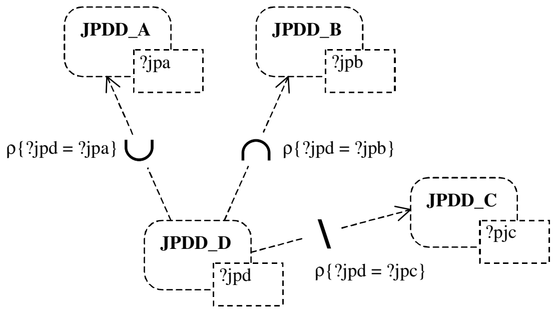


Fig. 4. Graphical representation of combination relationships between query models

query model is "calling for" the inclusion). Consequently, the identifiers of the including query model are deemed to be visible.

It remains to note that if a JPDD calls for multiple unions, confinements, and exclusions at the same time, combining is accomplished in the following order "1. all unions – 2. all confinements – 3. all exclusions". Any other combination order needs to be explicitly modeled using an execution tree.

4.2 Union

The first kind of relationship considers the selection criteria of the included (supplier) query model to be *alternative selection criteria* that extend those given in the including (client) query model. Consequently, the elements being designated by the including (client) query model are complemented with those being designated by the included (supplier) query model. Basically, this lead to a union of the designation results of both query models. Therefore, we use a union symbol "$\cup$" to symbolize this kind of relationship (see Fig. 4 for an example). In union relationships, mapping rules must be specified at least for all identifiers contained in the parameter box of the including (client) query model so that no element may be left unspecified in any tuple of the final designation result.

4.3 Confinement

The second kind of relationship looks at the selection criteria of the included (supplier) query model as *additional restrictions* that confine the selection criteria of the including (client) query model. Confinement is accomplished using mapping rules which are attached to the relationship and that indicate what identifier from the

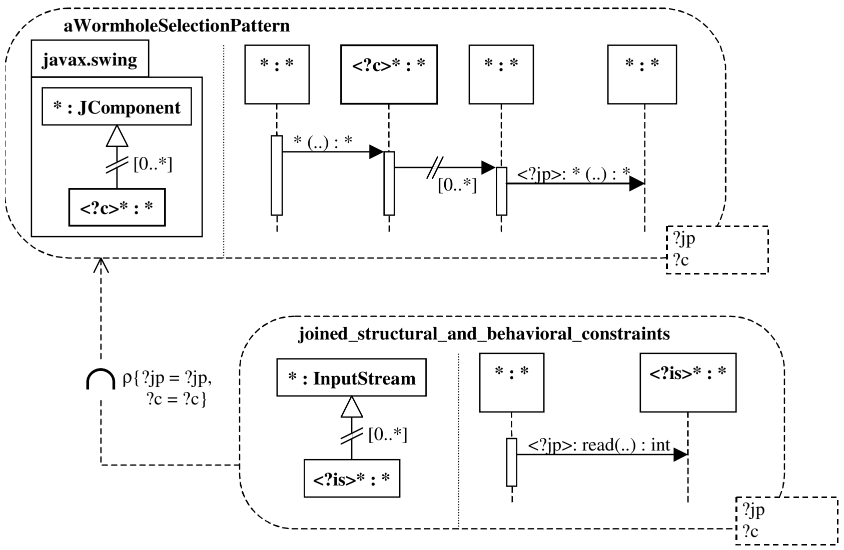


Fig. 5. Shorthand for concatenating selection constraints of different nature using a dotted line

including (client) query model confines what identifier from the included (supplier) query model. In consequence, the designation result of the including (client) query model is diminished to those tuples only that have a corresponding tuple in the designation result of the included (supplier) query model. We use an intersection symbol "∩" to indicate this kind of relationship (see Fig. 4 for an example).

As a short hand, confinement relationships can be abbreviated in a single query model by means of a dotted line. This is particularly useful when selection constraints of different nature (e.g. structural and behavioral selection constraints) are to be confined, and no distinct representation of either (or both) selection constraint(s) seems befitting. For example, Fig. 5 demonstrates how the structural and behavioral selection constraints from Fig. 3 (see section 3) can be merged into two rather than four query models. Further merging (e.g. in order to get one single all-inclusive query model) would be possible. Doing so, however, would obstruct the easy recognition of the application of the wormhole selection pattern in the query model.

4.4 Exclusion

The third kind of relationship regards the selection criteria of the included (supplier) query model to be *exclusion constraints*, which means that all tuples designated by the including (client) query model are excluded from its designation result if there is a corresponding tuple in the designation result of the included (supplier) query model. Again, correspondence between tuples is established by means of mapping rules that specify what identifier from the including (client) query model matches with what identifier from the included (supplier) query model. We use the difference symbol "\" to designate this kind of relationship (see Fig. 4 for an example).

4.5 Example

Having introduced the different combination relationships in detail, we now want to demonstrate their usage and their different effects with help of a small example.

Fig. 6 shows two query models, one of which is already well known from the previous examples (the bottom one). The other (top) one represents a general abstraction over recurring method calls: It selects two messages `<?jp1>` and `<?jp2>`, which both are sent to the same object `<?obj>`, and which both invoke methods with the (same) signature `<?sig>`. Message `<?jp1>` is "chained" to message `<?jp2>` with help of an indirect message symbol, which designates that message `<?jp2>` occurs in the control flow of message `<?jp1>`. Assuming that an object does not have two methods with the same signature, this query model selects all (equivalent) recurrences of a given message in its own control flow. This selection pattern is often needed to ensure that a given adaptation (or transformation) is applied only once (e.g. at the first occurrence of a method call) rather than every time the method is invoked.

The upper (supplier) query model is now included into the lower (client) query model in three different ways: ① by means of a confinement relationship, ② by means of a exclusion relationship, and ③ by means of a combination of both. The confinement relationships are annotated with mapping rules stating that message

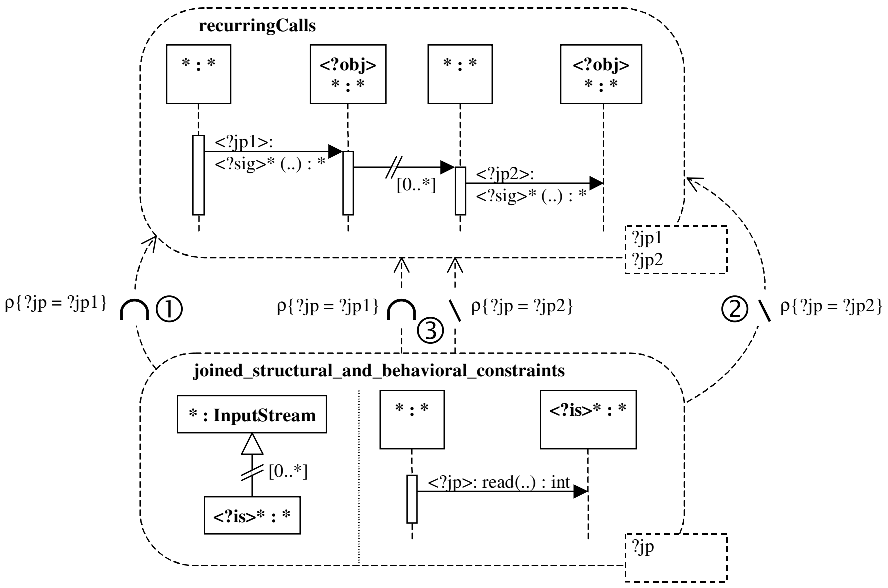


Fig. 6. Different usages of combination relationships

`<?jp>` from the lower (client) query model is mapped to message `<?jp1>` in the upper (supplier) query model. The exclusion relationships are annotated with mapping rules stating that message `<?jp>` from the lower (client) query model is mapped to message `<?jp2>` in the upper (supplier) query model. The effects of these combination relationships are as follows:

In the first case ①, method calls matching the lower (client) query model are only selected if there exists an recurring, i.e. an equivalent, method call in their control flow. As a result, all recursive method calls are selected, but the last. In the second case ②, method calls matching the lower (client) query model are selected only if they do not represent a recurring method call in the control flow of an equivalent method call. In consequence, only the first method call of a recurring set of equivalent method calls is selected, as well as any non-recurring method call. In the third case ③, the selection semantics of ① and ② are combined. As a result, all method calls representing the first ones in a recurring set of equivalent method calls are selected (due to the exclusion relationship); unlike to the previous case, though, selection of non-recurring method calls is not considered (due to the confinement relationship).

5 Semantics

In this section, the semantics of the relationships (that have been introduced in the previous section) are specified using the Object Constraint Language (OCL) 2.0 [19]. The OCL is chosen as it has been commonly proposed as the standard query language for QVT model transformations (cf. [10], [21], [6]).

Table 1. OCL semantics of combination relationships

-- identifier result sets
let QueryA_identifierResultSet(someModel : Namespace) :
    Set(TupleType(id_{A1} : $idType_{A1}$, ..., id_{Ai} : $idType_{Ai}$, ..., id_{Ak} : $idType_{Ak}$, ..., id_{An} : $idType_{An}$)) = ...
let QueryB_identifierResultSet(someModel : Namespace) :
    Set(TupleType(id_{B1} : $idType_{B1}$, ..., id_{Bj} : $idType_{Bj}$, ..., id_{Bl} : $idType_{Bl}$, ..., id_{Bm} : $idType_{Bm}$)) = ...
-- exposed result set
let QueryB_parameterResultSet(someModel : Namespace) :
    Set(TupleType(par_{B1} : $parType_{B1}$, ..., par_{Br} : $parType_{Br}$, ..., par_{Bs} : $parType_{Bs}$)) =
  QueryB_identifierResultSet(someModel)->collect(tup |
      Tuple{par_{B1} = tup.id_{B1}, ..., par_{Br} = tup.id_{Bj}, ..., par_{Bs} = tup.id_{Bm}})
-- result sets to be combined
let T1(..) = QueryA_identifierResultSet(..)
let T2(..) = QueryB_parameterResultSet(..)

-- set of tuple type labels in T1 and T2 having identical names (cf. [3])
let D = p.first.allAttributes->intersection(p.second.allAttributes)
-- set of joinable tuples from T1 $\times$ T2
let X(..) = T1(..)->product(T2(..))->select(p | D->forAll(d | p.first.d = p.second.d))

-- union
T1(..)->union(T2(..))
-- confinement
T1(..)->select(q | X(..)->exists(p | p.first = q))
-- exclusion
T1(..)->reject(q | X(..)->exists(p | p.first = q))

5.1 General OCL Semantics

Using the Object Constraint Language (OCL), the combination relationship can be defined more rigorously. To do so, we assume that each query model selects a set of model elements – i.e. those model elements that are given an identifier in the query model – from a given user-model. The model elements are returned as a set of tuples, while each tuple represents a distinct combination of model elements (from the given user-model) that satisfies the selection criteria outlined in the query model. In Table 1, these sets are rendered by `QueryA_identifierResultSet` and `QueryB_identifier-ResultSet`. The operations take one parameter (`someModel`) that is the user-model which is to be queried.

From all model elements in the query model that are given an identifier, only a projection is exposed for further processing. These model elements (i.e. their identifiers) are listed in the query model's parameter box. Again, the model elements are returned as a set of tuples. In Table 1, these set operations are exemplified by `QueryB_parameterResultSet`. And again, the operations take one parameter (`someModel`) identifying the user-model that is to be queried.

Available to a «union», «exclusion», or «confinement» combination relationship are (a) *all* identified model elements of the including (client) query model (i.e. `QueryA_identifierResultSet` in Table 1, subsequently referred to as `T1`)[2], as well as (b) *only the exposed* model elements of the included (supplier) query model (i.e.

QueryB_parameterResultSet in Table 1, subsequently referred to as T2)[2] (cf. Mapping Rules section).

In case of a union, the set of tuples T1 and T2 are unified using the OCL core operation union (see Table 1).

In case of a confinement, all pairs of tuples from T1 and T2 that comply to each other in all of their attributes having the same name (referred to as D in Table 1) are collected from the Cartesian product of T1 and T2 (following the approach of [3] to compute a relational join). These pairs of tuples (referred to as X in Table 1)[2] are then used to diminish the set of tuples T1 to those that are *also* part of X.

In case of an exclusion, initial proceeding is the same as in case of a confinement. Afterwards, though, the set of tuples T1 is diminished to those tuples that are *not* part of X (see Table 1).

Note that the semantics of both confinement and exclusion relationships are defined such that they allow the combination of query models whose result sets are different in structure (i.e. that comprise different – that is, only partially overlapping – sets of attributes). Note further that in the OCL code (see Table 1) we abstract from the mapping rules. Mapping, i.e. renaming of and projection to relevant identifiers in T1 and T2, cannot expressed trivially – and in a general way (!) – in the OCL (cf. [3], [4]). Hence, this can only be done manually for a particular combination relationship (as it will be demonstrated in the following).

5.2 A Concrete Example

After describing the general OCL semantics of the combination relationships in Table 1, we now take a look at their actual implications to a concrete example. We do so by revisiting the example from section 4.5.

First of all, we need to specify the relevant result sets of the query models, i.e. the identifier result set of the including query model as well as the parameter result set of the included query model (see above). Table 2 exemplifies how this is accomplished for query model recurringCalls[3] – referred to as QueryB in Table 2 (the specification of the result set of query model joined_structural_and_behavioral_constraints – referred to as QueryA in Table 2 – is omitted here for space reasons): The OCL code[4] starts out with collecting all possible combinations of model elements from the user-model being passed (someModel) that are of same type as the identifier model elements in the query model. In this example, the result set therefore contains all combinations of two stimuli model elements (stim1 and stim2), one instance model element (inst), and one operation model element (sig) that can be found in the user-model. Then, in a second step, this result set is reduced to only those combinations that consist of elements complying to the selection criteria specified in the query model. This evaluation is accomplished by special *matchModelElement* operations,

[2] Note that we abstract from parameter someModel of the result sets T1, T2, and X in the subsequent text and in Table 1.

[3] The approach has been inspired by the OCL code in [13].

[4] We used OCLE 2.02 (http://lci.cs.ubbcluj.ro/ocle) to syntax and type check the OCL code. Note that we abstract from any type cast (oclAsType) in the code shown.

Table 2. Applying the OCL semantics to the example

```
-- identifier result sets
let QueryA_identifierResultSet(someModel : Namespace) :
        Set(TupleType(stimulus : Stimulus, instance: Instance)) = [...]

let QueryB_identifierResultSet(someModel : Namespace) : Set(TupleType(stimulus1 : Stimulus,
stimulus2 : Stimulus, instance : Instance, signature : Operation)) =
-- create Cartesian product of all stimuli, instances, and signatures in someModel
someModel.allContents->select(me | me.oclIsKindOf(Stimulus))->collect(stim1 |
  someModel.allContents->select(me | me.oclIsKindOf(Stimulus))->collect(stim2 |
    someModel.allContents->select(me | me.oclIsKindOf(Instance))->collect(inst |
        someModel.allContents->select(me | me.oclIsKindOf(Operation))->collect(sig |
    Tuple {
        stimulus1 : Stimulus = stim1,
        stimulus2 : Stimulus = stim2,
        instance : Instance = inst,
        signature : Operation = sig
    } ))))
-- select those tuples that match the selection criteria specified in the query model...
->select(tup | tup.stimulus1.matchesStimulus(self.allContents->any(me | me.name="jp1"))
    and tup.stimulus2.matchesStimulus(self.allContents->any(me | me.name="jp2"))
    and tup.instance.matchesInstance(self.allContents->any(me | me.name="obj"))
    and tup.signature.matchesOperation(self.allContents->any(me | me.name="sig"))
-- ...and whose elements are related to each other as specified in the query model
    and tup.stimulus1.receiver = tup.instance
    and tup.stimulus2.receiver = tup.instance
    and tup.stimulus1.dispatchAction.method.specification = tup.signature
    and tup.stimulus2.dispatchAction.method.specification = tup.signature
    and tup.stimulus2.allActivators()->includes(tup.stimulus1))

-- exposed result set
let QueryB_parameterResultSet(someModel : Namespace) :
        Set(TupleType(stimulus1 : Stimulus, stimulus2 : Stimulus)) =
-- project identifier result set to those elements being exposed by the query model
   QueryB_identifierResultSet(someModel)->collect(tup |
        Tuple{ stimulus1 = tup.stimulus1, stimulus2 = tup.stimulus2 })

-- result sets to be combined
let T1(someModel : Namespace) = QueryA_identifierResultSet(someModel)
let T2(someModel : Namespace) = QueryB_parameterResultSet(someModel)
-- set of joinable tuples from T1 × T2 (according to mapping rules outlined in Fig. 6)
let X1(someModel : Namespace) = T1(someModel)->product(T2(someModel))
  ->select(p | p.first.stimulus = p.second.stimulus1)
let X2(someModel : Namespace) = T1(someModel)->product(T2(someModel))
  ->select(p | p.first.stimulus = p.second.stimulus2)

-- confinement (①)
let ResultSet_1(someModel : Namespace) = T1(someModel)
  ->select(q | X1(someModel)->exists(p | p.first = q))
-- exclusion (②)
let ResultSet_2(someModel : Namespace) = T1(someModel)
  ->reject(q | X2(someModel)->exists(p | p.first = q))
-- simultaneous confinement and exclusion (③)
let ResultSet_3(someModel : Namespace) = T1(someModel)
  ->select(q | X1(someModel)->exists(p | p.first = q))
  ->reject(q | X2(someModel)->exists(p | p.first = q))
```

such as they have been specified in [23] [22]. The operations take as parameter an identifier model element from the query model (i.e. `jp1`, `jp2`, `obj`, and `sig`, respectively[5]); the parameter is taken as a selection pattern to which the user-model elements are compared. At last, the OCL code assures that the elements in the remaining combinations actually relate to each other as specified in the query model (the evaluation is based on the (meta-)associations between the model elements as specified in the UML meta-model [20]).

Once we have retrieved the identifier result set of the (included) query model (`QueryB_identifierResultSet`), we need to project that result set such that its tuples only consists of model elements that are actually exposed by the query model. Therefore, a parameter result set (`QueryB_parameterResultSet`) is created from the identifier result set, whose tuples only consist of model elements that are (designated by the identifier model elements) listed in the parameter box of the query model (i.e. the stimuli `stimulus1` and `stimulus2`, in this case).

Now we are ready to perform the confinement and the exclusion, respectively. To do so, two sets `X1` and `X2` of "joinable tuples" are created – one for each mapping specification given in the example (see Fig. 6). The sets comprise all pairs of tuples from `QueryA_identifierResultSet` (referred to as `T1`) and `QueryB_parameter-ResultSet` (referred to as `T2`) that are referring to the same model elements in their attributes being mapped (i.e. in `stimulus` and `stimulus1`, as well as in `stimulus` and `stimulus2`, respectively). The sets are then used to filter the identifier result set of the including query model (`QueryA_identifierResultSet`, referred to as `T1`) such that it consists only of those tuples ① being part of `X1` (resulting into a confinement), ② being *not* part of `X2` (resulting into an exclusion), and finally, ③ being part of `X1`, but not part of `X2` (resulting into a combined confinement and exclusion).

6 Related Work

JPDDs as we have used them in this paper relate to other approaches that provide visualization means for model queries in the MDA domain. Examples of such approaches are basically all proposals to specify model transformations, such as MOLA [11], BOTL [17], or the QVT-Merge submission [21], etc. However, none of these approaches provide means to reason about model queries in isolation. In consequence, no abstraction means are provided to segregate recurring selection patterns, and no relationships are specified to re-use such recurring selection patterns in different discrete application domain-specific selection queries.

Instead, OCL 2.0 is commonly suggested to serve as a (purely textual) query language – e.g. for OMG's transformation language QVT (cf. [10], [21], [6]). Therefore, it is interesting to relate our combination relationships to OCL's proper capabilities to express and calculate combination of sets of tuples. Several studies [16] [1] [3] have been conducted in this regard to investigate the expressiveness of OCL with respect to relational algebra [5], a well-known and well-founded approach to

[5] It is assumed that the OCL code is specified in the context of the query model; thus, `self.allContents` refers to all model elements contained in `QueryB`. See [22] for further information on how identifiers and name patterns are stored in the meta-representation of model elements.

specify operations on (e.g. combinations of) sets of tuples, originating from the database domain. In the latest study [3], it has been stated that since the introduction of tuple types and product types as primitive operators in OCL 2.0, OCL became much more expressive. But still, projection and renaming of tuple attributes needs to be done on a per-case basis and cannot be expressed in a general way. When compared to the mentioned work, it is important to note that our work is not focused on providing a (relational) complete set of graphical operators that could serve as a basis for arbitrary computations over sets of tuples (of model elements). Instead, the goal was to find appropriate abstractions for common combinations of query models and daily needs in query design.

Nonetheless, the identified combination relationships can be compared to relational operations, and would correspond to the following expressions: Assuming that sets of tuples `T1` and `T2` (see Table 1) designate relations, a union relationship equates to the following relational operation: $T1 \cup T2$. A confinement relationship could be expressed as a left semi-join between `T1` and `T2`: $T1 \ltimes T2$. And finally, an exclusion relationship equates to $T1 - (T1 \ltimes T2)$. A mapping rule serves two purposes: First of all, it projects the second relation `T2` to those attributes that are relevant for the respective relational operation: $\pi_{attT2.1,\ attT2.2,\ ...,\ attT2.n}\ T2$. Furthermore, it renames the projected attributes so that their labels comply to the attribute labels of the first relation `T1` (and the relational operation can actually be performed): $\rho_{attT1.1 \leftarrow attT2.1,\ attT1.2 \leftarrow attT2.2,\ ...,\ attT1.n \leftarrow attT2.n}$

Apart from that, the presented relationships also compare to (and have been influenced by) the combination operators for join point queries (so-called "pointcuts") `&&`, `||`, and `!` in AspectJ [14], the most popular aspect-oriented programming language. If two join point queries are combined by means of an AND-operator `&&`, the result set contains all join points that are picked out by *both* queries – which corresponds to the selection semantics of the confinement relationship. If two join point queries are combined by means of an OR-operator `||`, the result set contains all join points that are picked out by *either* query – which equates to the union relationship. If a join point query is prepended by a NOT-operator `!`, the join points designated by that query are dismissed from the selection result. This complies to the exclusion relationship (that is, to be more exact, the exclusion relationship equates to a combination of an AND-operator and a NOT-operator in AspectJ, i.e. "`&& !`").

7 Conclusion

In this paper we elucidated the need of having suitable means to combine query models. We presented three kinds of combination relationships – «union», «confinement», and «exclusion» – and detailed their semantics informally and with help of OCL 2.0 expressions. We demonstrated the usage and the benefits of these relationships with help of two common selection patterns.

By means of the presented combination relationships, developers are able to abstract from recurring selection patterns and to reuse them in different application contexts. Furthermore, they are able to relate queries pertaining to different modeling notations or diagrams (e.g. to both structural and behavioral model specifications); thus, they are able to specify context-sensitive model queries. Of course, in order to efficiently do so in model transformations, the meta models of the involved modeling notations must share some common abstractions.

Provided with the combination relationships presented in this paper, abstractions of recurring selection patterns can be assembled in public libraries (just like abstractions of common transformations can be assembled in public transformation libraries), ready for reuse by the model-driven developer. That way, developers are freed from elaborating on recurring selection semantics again and again (e.g. how to handle recursive method calls; see section 4.5), each time they are dealing with a new problem.

It remains to mention that our work is not concerned with the actual evaluation of query combinations. That is, we do not consider how the resulting sets of tuples (of model elements) are actually extracted from a given user model. In particular, we currently abstract from problems that may occur due to recursive or circular query combinations. Such combinations may lead to infinite loops during query evaluation, or may introduce irresolvable dependencies between the involved query specifications. It is an interesting field for future research to investigate if such problems may be detected prior to query execution time and how they could be possibly resolved automatically based on static query analysis.

References

[1] Akehurst, D., Bordbar, B., *On Querying UML Data Models with OCL*, in: Proc. of UML'01, Toronto, Canada, LNCS 2185, Springer, October 2001, pp. 91-103

[2] Aßmann, U., Aksit, M., Rensink, A., *Model Driven Architecture* (European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004; Revised Selected Papers), LNCS 3599, Springer, June 2004

[3] Balsters, H., *Modelling Database Views with Derived Classes in the UML/OCL-Framework*, in: Proc. of UML'03, San Francisco, CA, LNCS 2863, Springer, October 2003, pp. 295-309

[4] Blaha, M., Premerlani, W., *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, Englewood Cliffs, NJ, 1998

[5] Codd, E.F., *Relational Completeness of Data Base Sublanguages*, in: Rustin, R. (ed.), Courant Computer Science Symposia, Vol. 6: Database Systems, Prentice Hall, Englewood Cliffs, NJ, 1972, pp. 65-98

[6] Compuware Corporation, SUN Microsystems, *2nd revised submission for MOF 2.0 Query/Views/Transformations RFP*, 11. October 2004 (OMG Document ad/2004-10-03)

[7] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, 1995

[8] Hanenberg, S., Schmidmeier, A., *Idioms for Building Software Frameworks in AspectJ*, 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), Boston, MA, March 17, 2003

[9] Hannemann, J., Kiczales, G., *Design pattern implementation in Java and AspectJ*, in: Proc. of OOPSLA'02, Seattle, Washington, SIGPLAN Notices 37(11), ACM, November 2002, pp. 161-173

[10] Interactive Objects Software, Project Technology, *2nd revised Submission for MOF 2.0 Query/Views/Transformations RFP*, 12. January 2004 (OMG Document ad/2004-01-14.pdf)

[11] Kalnins, A., Barzdins, J., Celms, E., *Model Transformation Language MOLA*, in: [2], pp. 62-76

[12] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, Chr., Lopes, C.V., Loingtier, J-M., Irwin, J.: *Aspect-Oriented Programming*, in: Proc. of ECOOP '97, Jyväskylä, Finland, LNCS 1241, Springer, June 1997, pp. 220-242

[13] Kožusznik, J., *Dotazování, pohledy a transformace v MDA*, in: Proc. of Objekty 2003, Ostrava, Czech Republic, ISBN 80-248-0274-0, VŠB-Technical University of Ostrava, November 2003, pp. 120-128

[14] Laddad, R., *AspecJ in Action: Practical Aspect-Oriented Programming*, Manning Publications, Greenwich, 2003

[15] Lesiecki, N., *Enhance design patterns with AspectJ*, Part 1, IBM DeveloperWorks > Java Technology > AOP@Work (http://www-128.ibm.com/developerworks/java/library/j-aopwork5)

[16] Mandel, L., Cengarle, M., *On the Expressive Power of OCL*, in: Proc. of FM'99, Toulouse, France, LNCS 1708, Springer, September 1999, pp. 854-874

[17] Marschall, F., Braun, P., *Model Transformations for the MDA with BOTL*, in: Workshop Proc. of MDAFA 2003, Enschede, The Netherlands, CTIT Technical Report TR-CTIT-03-27, University of Twente, June 2003, pp. 25-36

[18] OMG, *MDA Guide Version 1.0*, 2003 (OMG Document omg/2003-05-01)

[19] OMG, *OCL 2.0 Final Adopted Specification*, 2003 (OMG Document ptc/03-10-14)

[20] OMG, *Unified Modeling Language Specification*, Version 1.5, 2003 (OMG Document formal/03-03-01)

[21] QVT-Merge Group, *Revised submission for MOF 2.0 Query / Views / Transformations RFP*, 2. March 2005 (OMG Document ad/2005-03-02)

[22] Stein, D., Hanenberg, S., Unland, R., *A Graphical Notation to Specify Model Queries for MDA Transformations on UML Models*, in: [2], pp. 77-92

[23] Stein, D., Hanenberg, S., Unland, R., *Query Models*, in: Proc. of UML'04, Lisbon, Portugal, LNCS 3273, Springer, October 2004, pp. 98-112

Transformations Between UML and OWL-S

Roy Grønmo[1], Michael C. Jaeger[2], and Hjørdis Hoff[1]

[1] SINTEF Information and Communication Technology,
P.O.Box 124 Blindern, N-0314 Oslo, Norway
{Roy.Gronmo, Hjordis.Hoff}@sintef.no
[2] Technische Universität Berlin, Institute of Telecommunication Systems,
SEK FR6-10, Franklinstrasse 28/29, D-10587 Berlin, Germany
mcj@cs.tu-berlin.de

Abstract. As the number of available Web services increases there is a growing demand to realize complex business processes by combining and reusing available Web services. The reuse and combination of services results in a composition of Web services that may also involve services provided in the Internet. With semantically described Web services, an automated matchmaking of capabilities can help identify suitable services. To address the need for semantically defined Web services, OWL-S and WSML have been proposed as competing semantic Web service languages.

Both proposals are quite low-level and hard to use even for experienced Web service developers. We propose a UML profile for semantic Web services that enables the use of high-level graphical models as an integration platform for semantic Web services. The UML profile provides flexibility as it supports multiple semantic Web service languages. Transformations of both ways between OWL-S and UML are implemented to show that the UML profile is expressive enough to support one of the leading semantic Web service languages.

1 Introduction

A growing number of Web services are implemented and made available internally in an enterprise or externally for other users to invoke. These Web services can be reused and composed in order to realize larger and more complex business processes. We define *Web services* to be services made available by using Internet protocols such as HTTP and XML-based data formats for their description and invocation. Web service registries such as UDDI allows for Web services to be published and discovered. A *Web service composition* is a description of how Web services can interoperate in order to perform a larger task.

The Web service proposals for description (WSDL [16]), invocation (SOAP [15]) and composition (BPEL4WS [2][1]) that are most commonly used, lack proper semantic descriptions of services. This makes it hard to search for appropriate services because a large number of syntactically described services need to be manually interpreted to see if they can perform the desired task. The requester may also need additional communication with the provider to determine the suitability of the service.

Semantically described Web services make it possible to improve the precision of the search for existing services and to automate the composition of services. Two recent

[1] BPEL4WS is currently being updated. It will be released in future under the name *WS-BPEL.*

proposals have gained a lot of attention; The American-based OWL Services (OWL-S) [4] and the European-based Web Services Modeling Language (WSML[2]) [17]. These emerging specifications overlap in some parts and are complementary in other parts. They are both described by low-level lexical notations. WSML uses its own lexical notation, while OWL-S is XML-based.

The leading organization for object-oriented programming, the Object Management Group (OMG), has defined the Unified Modeling Language (UML), a standard graphical language for expressing system development models. OMG also promotes a Model-Driven Architecture (MDA) approach for analysis, design and implementation of software systems. In a model-driven development process, models are used to describe business concerns, user requirements, information structures, components and component interactions. These models govern the system development because they can be transformed into executable code.

The work described in this paper adopts the MDA strategy for developing compositions of semantic Web services. An important question to be addressed in this paper is: *How can the new semantic Web service proposals be utilized within a model-driven Web service composition methodology?* An important part of the main question is to investigate if UML 2.0 [11] can be used as an integration platform for modeling semantic Web services. This implies the following requirements for the resulting UML models:

1. *Expressiveness.* They can contain sufficient semantic annotations so that they can be transformed to and from complete semantic Web service documents.
2. *Independence.* They are independent of the lexical semantic Web service languages.
3. *Readability.* They are easy to understand, interpret and specify for experienced modelers.

The motivation of the independence requirement is that one does not want to be tied to a particular semantic language, especially when there are competing standards. Furthermore, we believe that the semantic Web service developer can become more productive when working at a higher level than the low-level XML code. In order to satisfy the requirements, we have suggested a UML profile for semantic Web services and implemented transformations for both ways between UML and OWL-S. In addition, we explain how our UML profile can be transformed to WSML description, which demonstrates its semantic Web language-independence.

The paper is structured as follows: Section 2 briefly describes the procedure for semantic Web service composition that clarifies the need for a UML profile and transformations between UML and semantic Web service languages; Section 3 introduces the UML profile for semantic Web services; Section 4 presents the transformation rules both ways between OWL-S and UML; Section 5 shows the correspondence between WSML, OWL-S and our UML Profile; Section 6 evaluates our approach; Section 7 covers related work; and finally Section 8 concludes the paper.

[2] Note that WSML is often referred to as WSMO as it is defined by the WSMO working group and the WSML has only recently been published.

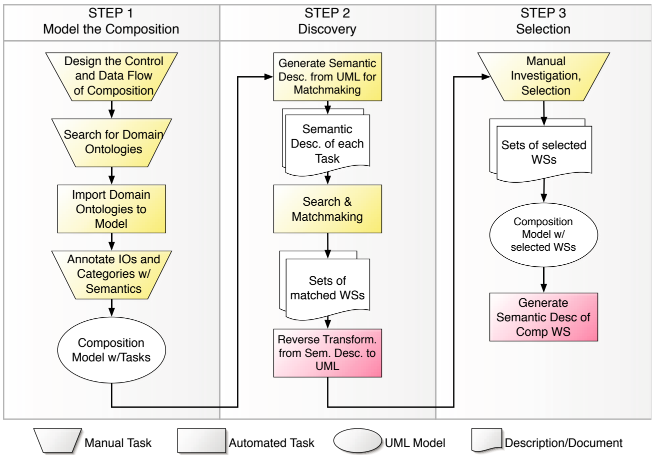

STEP 1
Model the Composition
STEP 2
Discovery
STEP 3
Selection
Design the Control and Data Flow of Composition
Search for Domain Ontologies
Import Domain Ontologies to Model
Annotate IOs and Categories w/ Semantics
Composition Model w/Tasks
Generate Semantic Desc. from UML for Matchmaking
Semantic Desc. of each Task
Search & Matchmaking
Sets of matched WSs
Reverse Transform. from Sem. Desc. to UML
Manual Investigation, Selection
Sets of selected WSs
Composition Model w/ selected WSs
Generate Semantic Desc of Comp WS
Manual Task
Automated Task
UML Model
Description/Document

Fig. 1. Procedure for Composition of Semantic Web Services

2 Model-Driven Semantic Web Service Composition

This Section presents the model-driven procedure for creating compositions of semantically described Web services (Figure 1). The procedure consists of three main steps. The first step focuses on modeling a composition consisting of tasks, assuming that specific services are not yet identified. The second step focuses on service discovery and matchmaking based on the semantic description of tasks and services. In the third and final step, the service candidates are selected and the composition is completed. The procedure is described in a sequential manner although it should be emphasized that the procedure is an iterative process.

Step 1. The modeler specifies a new Web service composition in UML by creating a model that defines the control and data flow between the tasks. Each task is given with a task name and required inputs and outputs. Then, the designer must identify appropriate domain ontologies to semantically annotate the UML model. Domain ontologies may be searched at organizations or business interest communities. For example, the North American Industry Classification System represents such an effort[3] and provides an ontology of products and services. After the modeler has determined one or more candidate ontologies, he imports these into the UML model. Such a technique has already been proposed by the work of Duric to bring OWL ontologies into a UML modeling environment [1]. Then, the desired Web services can be classified with inputs and outputs and a service category.

Only for the case that appropriate ontologies do not exist or are not suitable for being extended, the designer should consider to design a new ontology. However, we do not expect the need for creating a new ontology for the following reason: if a service

[3] The ontology is available at http://www.naics.com/. An OWL version of this ontology is provided at http://www.daml.org/ontologies/.

requester cannot find ontologies, then he cannot presume that a provider provides a semantic description of his services using those non-existing ontologies. The outcome of the first step is a composition model that contains all the needed information for performing a discovery of service candidates.

Step 2. The second step handles discovery of suitable Web services. The discovery process is based on a matchmaking algorithm for semantic descriptions of services. It is assumed that a Web service registry is available with the following information provided for each Web service: a) a service interface description with location and b) a semantic description that can be used for the matchmaking process. The abstract composition model produced in step 1 is automatically transformed into a lexical document ("Semantic Desc. of each Task" in Figure 1) that can be parsed by a search and matchmaking algorithm. The recent proposals for matchmaking algorithms deal only with semantic matching of inputs, outputs and categories [9]. However, even if all these parts have clear matches, we are not guaranteed that the provided service is a perfect match. To guarantee perfect matches, further reasoning would be necessary that also take preconditions, postconditions and effects into account.[4] Since this topic requires a lot of research, software tools and adoption of such techniques by the user community, we do not expect that all services can be discovered fully automatic in the near future. The matchmaking of inputs, outputs and categories will improve the precision of the search, but it will not remove the need for manual investigation of the discovered Web services to determine if they are suitable or not.

We anticipate that the found candidates provide a semantic description of their capabilities. The leading proposals for such semantic descriptions are OWL-S and WSML. However, the low-level and verbose OWL-S or WSML files are time-consuming and demanding to comprehend for the designer. To ease the manual investigation process, we propose to *reverse engineer* OWL-S and WSML into high-level UML models. Another benefit of importing the semantic description into UML, is that the imported services can be used directly as UML elements when finalizing the composition model. To accomplish this step, we have realized a transformation from OWL-S to UML which is further described in Section 4. The outcome of step 2 is a set of candidate services for each task.

Step 3. Based on the reverse engineered semantic description, the investigation of the services in UML can take place. The modeler selects the appropriate services and ideally at least one chosen service is assigned to each task. Tool support will preferably help the modeler to finalize the concrete composition model, which is the outcome of step 3. Theoretically, it could be wise to choose more than one service for a task. If during run-time a service becomes temporarily or permanently unavailable, an alternative service performing the same task can compensate the unavailable one.

At last, the concrete composition model is used to generate a semantic Web service description of the newly composed service (OWL-S, WSML etc.). This description can

[4] The OWL-S proposal considers only the element *effects* to define resulting conditions from the service execution. The WSML proposal features both: it declares that postconditions cover the data output while effects can be used to describe general state changes not covering to the output.

be automatically generated by the transformation which we have explained further in Section 4. The generated lexical semantic Web service descriptions can be published in global registries which enables third parties to discover and use the composed service.

3 UML Profile for Semantic Web Services

This section defines a UML profile which can be used to model semantic aspects of Web services. UML offers three extension mechanisms that can be used when extending UML's base meta-model elements:

- *stereotypes* introduce new elements by extending a base element,
- *tagged values* introduce new properties for the new element, and
- *constraints* are restrictions on the new element with respect to its base element.

A group of UML extensions constitute a UML profile. Figure 2 shows a meta-model of our proposed UML profile. We build the UML profile on top of two existing meta-models which are shown as packages in the figure. The first reused meta-model is the UML Ontology Profile that is most relevant when defining associations to our UML profile. UOP which is defined by Duric [1]. UOP is an extension to UML 1.5 class models which we have upgraded to UML 2.0. UOP captures ontology concepts with properties and relationships. The UOP package in the Figure contains the two elements (Ontology and OntClass. An OntClass element extends a UML class and represents a semantic concept. The OntClasses are semantically defined types which are grouped into Ontology packages. By modeling the semantic concepts in UOP, there will be a set of UML elements available to use as semantic types in the to be defined semantic Web service composition models.

The second meta-model represents standard UML 2.0 activity model elements. Our UML profile is defined as an extension to UML 2.0 activity models by extending the elements shown inside the standard UML 2.0 package. The new UML profile elements are shown in a darker color than the extended and reused elements. In order to avoid a cluttered diagram, we intentionally do not show tagged values, constraints, and the associations of the four elements Pre-condition, Post-condition, Category and Effect. These omitted relationships are explained in the subsequent paragraph.

The most central concept in the profile is WebService which extends the UML Activity element. A WebService represents a single callable Web service operation,

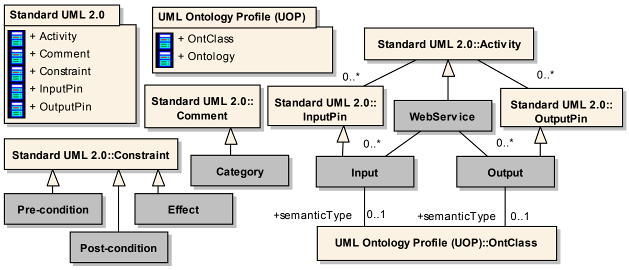


Fig. 2. The Meta-model of the UML profile

Table 1. Summary of the UML profile

Stereotype	Extending UML Meta-model Element	Tagged Values	Usage
WebService	Activity	wsdl, service, port, operation	Used to model a single Web service operation. It's tagged values are sufficient to identify a web service operation
input	Activity inputPin		The stereotype is added to visualize if a pin is an input or an output. This is implicit when the WebService is part of a composition with incoming and outgoing flows, but is not visible when the WebService is visualized as a single service. It has an optional semantic type given by an association to a UOP OntClass.
output	Activity outputPin		<Same as for input>
Category	Comment	taxonomy, taxonomyURI, value, code	This item links the WebService to a category defined by a semantically defined concept within an ontology.
Pre-condition	Constraint		A pre-condition is of Constraint type and is visualized by a note in the diagram. A pre-condition is attached to all the input parameters included in the pre-condition statement. If no input parameters are part of the statement, then it is attached to the WebService item. The boolean pre-condition statement must be satisfied in order to execute the WebService.
Post-condition	Constraint		<Same as for Pre-condition with respect to output parameters instead of input parameters.> The boolean post-condition statement must be true when the WebService has terminated its execution.
Effect	Constraint		An effect statement defines the result of executing the WebService. The effect item is attached to the WebService

as opposed to a collection of operations. A WebService has four tagged values (wsdl, service, port, operation) which uniquely identify a Web service operation within a WSDL file. The other extensions to UML 2.0 activity models are introduced to support semantic annotation of WebService elements. A WebService can have an arbitrary number of Input and Output parameters. The Input and Output elements are minor extensions to the inputPin and outputPin of a UML Activity. The type of each of the parameters can be a syntactic type as previously for standard UML 2.0 Pins, but preferably now it will be a semantic type given as a UOP OntClass.

A semantic categorization of the WebService is given by a link to a Category element. The Category element extends the UML Comment element. It must be linked to a WebService and it has four tagged values which identify a category concept defined within an ontology. Although pre-conditions and post-conditions already exist in the UML meta-model, we have introduced two new elements for this purpose by extending the UML Constraint element. If a Pre-condition includes more than one input parameter in its expression, then it will be linked to all of the included parameters. The same rule applies to Post-condition with respect to output parameters. If the pre- or post-condition does not refer to any parameter, then it must be linked directly to the WebService. Note that the semantics of these new pre- and post-conditions are the same as in those already present in UML 2.0 by being conditions that apply to an operation (in this case specialised to be a Web service operation). The improvement is that they are now clearly linked to the Pins it concerns for improving the visualisation of the UML diagrams. Finally, the Effect element extends the UML Constraint. It is linked to a WebService to indicate the result of a successful execution of the WebService. The contents of Pre-condition, Post-condition and Effect should all be boolean expressions where the Object Constraint Language (OCL) is a natural candidate. Table 1 contains a summary of all the new elements in our UML profile. When modeling in the

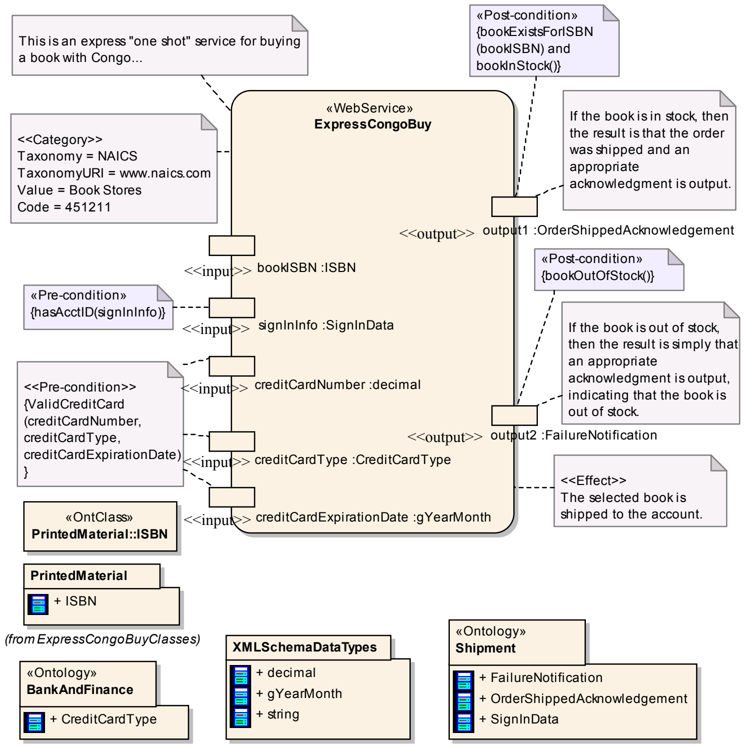

This is an express "one shot" service for buying a book with Congo...
«Post-condition» {bookExistsForISBN (bookISBN) and bookInStock()}
«WebService» ExpressCongoBuy
<<Category>> Taxonomy = NAICS TaxonomyURI = www.naics.com Value = Book Stores Code = 451211
If the book is in stock, then the result is that the order was shipped and an appropriate acknowledgment is output.
<<output>> output1 :OrderShippedAcknowledgement
<<input>> bookISBN :ISBN
«Post-condition» {bookOutOfStock()}
«Pre-condition» {hasAcctID(signInInfo)}
<<input>> signInInfo :SignInData
If the book is out of stock, then the result is simply that an appropriate acknowledgment is output, indicating that the book is out of stock.
<<input>> creditCardNumber :decimal
<<Pre-condition>> {ValidCreditCard (creditCardNumber, creditCardType, creditCardExpirationDate) }
<<output>> output2 :FailureNotification
<<input>> creditCardType :CreditCardType
<<Effect>> The selected book is shipped to the account.
«OntClass» PrintedMaterial::ISBN
<<input>> creditCardExpirationDate :gYearMonth
PrintedMaterial
+ ISBN
(from ExpressCongoBuyClasses)
XMLSchemaDataTypes
+ decimal
+ gYearMonth
+ string
«Ontology» Shipment
+ FailureNotification
+ OrderShippedAcknowledgement
+ SignInData
«Ontology» BankAndFinance
+ CreditCardType

Fig. 3. ExpressCongoBuy service represented in our UML profile

UML profile it is important to note that also the reused elements from the UOP and the UML 2.0 activity models are available.

Now that we have defined how to model single, semantically annotated Web services in UML, we use UML 2.0 activity models to model compositions of semantically annotated Web services. The built-in control and data flow capabilities allow us to define how single semantic Web services interoperate in order to accomplish larger tasks. The resulting composition model can have its own boundary definition with a set of input and output parameters and thus can expose itself as a new Web service. A Web service may be decomposed as a composition, with some of its Web services being compositions itself and this may be recursively continued at an arbitrary number of levels.

3.1 The ExpressCongoBuy Example Expressed in the UML Profile

The model instance in figure 3 is used to explain our proposed UML profile by showing the OWL-S reference example ExpressCongoBuy in UML.[5] ExpressCongoBuy is a Web service that allows a customer to buy books. In the ExpressCongoBuy example

[5] The Congo example used can be found at the OWL-S home page: http://www.daml.org/services/owl-s/1.1/examples.html.

there are five input parameters to identify the customer information (signInInfo), credit card (creditCardNumber, creditCardType and creditCardExpirationDate) and the book (bookISBN).

There are two mutually exclusive output parameters in the example.The first output parameter indicates that the book is successfully purchased and shipped to the buyer's address, while the second output provides a message informing that the book was out of stock. The parameter types are linked to syntactic and semantic types. In the ExpressCongoBuy example the parameters creditCardNumber and creditCardExpirationDate are syntactically defined by referring to standard XML Schema data types. The other parameters are defined with semantic types as UOP OntClasses and grouped inside UOP Ontology packages.

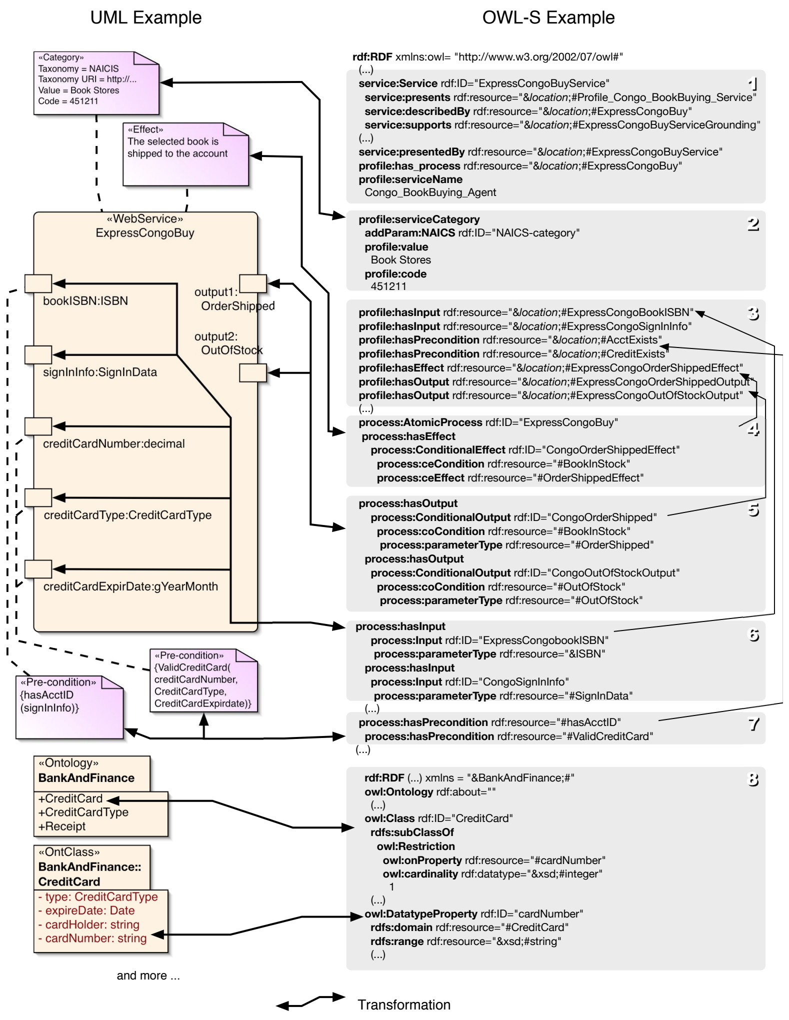
Note that the model in figure 3 has only pseudo-logical expressions as the content of the pre-conditions, post-conditions and effect elements to make the example more readable. The post-conditions are attached to the output pins which states that there is a conditional output parameter. Only one of the two output parameters is returned, depending on which of the two post-conditions that evaluates to true. Finally we encourage the extensive use of human-understandable comments as UML notes attached to the relevant UML elements. This will help the model reader to interpret the model. All the three used plain comments (without stereotypes) in the UML model example are the original comments as found in the OWL-S example document.

4 Transformations Between OWL-S and Our UML Profile

The transformation between OWL-S and UML is necessary for two tasks when building semantic Web service compositions: a) to facilitate the reengineering process using the UML view, and b)to publish the semantically annotated composed service at the end. This Section provides an overview of our proposed transformation rules for a) and b). Figure 4 shows a schematic view of the transformation elements. The figure consists of two parts: the left side shows the UML representation of the service. And the right side outlines fragments of the OWL-S description using a simplified non-XML-notation which is less verbose than the true XML. In between the two parts, the arrows indicate which part of the model corresponds to which part in OWL-S.

The service is represented in the UML model according to our UML profile as an activity with the stereotype WebService. Parameters of this activity element represent the inputs and outputs. The Web service has got five inputs and two (conditional) outputs. The figure also shows that properties of the service – such as the pre- and postcondition – are visualised with stereotyped notes. On the OWL-S side such an activity corresponds to the frame of one OWL-S document (indicated by the box labeled 1 in the figure). The inputs and outputs of an activity correspond to the hasInput and hasOutput elements in OWL-S accordingly (labelled 5 and 6).

In OWL-S it is proposed to use the Semantic Web Rule Language (SWRL [5]) for representing the hasPrecondition, hasPostcondition and hasEffect elements (labeled 4 and 7, please note that the figure does not show the postcondition due to space limitations). In UML, our proposal uses stereotyped notes containing an expression using OCL. However, the transformations of logical expressions would signifi-

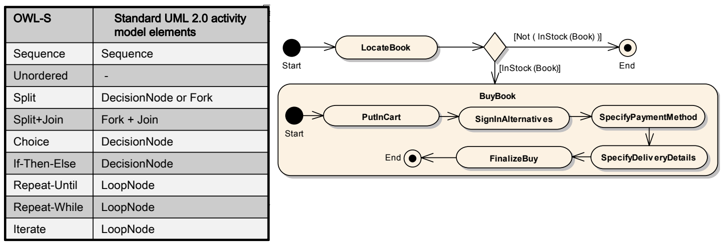

UML Example
OWL-S Example
«Category» Taxonomy = NAICS Taxonomy URI = http://... Value = Book Stores Code = 451211
«Effect» The selected book is shipped to the account
«WebService» ExpressCongoBuy
bookISBN:ISBN
signInInfo:SignInData
creditCardNumber:decimal
creditCardType:CreditCardType
creditCardExpirDate:gYearMonth
output1: OrderShipped
output2: OutOfStock
«Pre-condition» {ValidCreditCard(creditCardNumber, CreditCardType, CreditCardExpirdate)}
«Pre-condition» {hasAcctID (signInInfo)}
«Ontology» BankAndFinance
+CreditCard
+CreditCardType
+Receipt
«OntClass» BankAndFinance:: CreditCard
- type: CreditCardType
- expireDate: Date
- cardHolder: string
- cardNumber: string
and more ...
Transformation
1
2
3
4
5
6
7
8

```
rdf:RDF xmlns:owl= "http://www.w3.org/2002/07/owl#"
 (...)
 service:Service rdf:ID="ExpressCongoBuyService"
   service:presents rdf:resource="&location;#Profile_Congo_BookBuying_Service"
   service:describedBy rdf:resource="&location;#ExpressCongoBuy"
   service:supports rdf:resource="&location;#ExpressCongoBuyServiceGrounding"
 (...)
 service:presentedBy rdf:resource="&location;#ExpressCongoBuyService"
 profile:has_process rdf:resource="&location;#ExpressCongoBuy"
 profile:serviceName
   Congo_BookBuying_Agent

 profile:serviceCategory
   addParam:NAICS rdf:ID="NAICS-category"
   profile:value
     Book Stores
   profile:code
     451211

 profile:hasInput rdf:resource="&location;#ExpressCongoBookISBN"
 profile:hasInput rdf:resource="&location;#ExpressCongoSignInInfo"
 profile:hasPrecondition rdf:resource="&location;#AcctExists"
 profile:hasPrecondition rdf:resource="&location;#CreditExists"
 profile:hasEffect rdf:resource="&location;#ExpressCongoOrderShippedEffect"
 profile:hasOutput rdf:resource="&location;#ExpressCongoOrderShippedOutput"
 profile:hasOutput rdf:resource="&location;#ExpressCongoOutOfStockOutput"
 (...)
 process:AtomicProcess rdf:ID="ExpressCongoBuy"
  process:hasEffect
    process:ConditionalEffect rdf:ID="CongoOrderShippedEffect"
      process:ceCondition rdf:resource="#BookInStock"
      process:ceEffect rdf:resource="#OrderShippedEffect"

 process:hasOutput
    process:ConditionalOutput rdf:ID="CongoOrderShipped"
      process:coCondition rdf:resource="#BookInStock"
       process:parameterType rdf:resource="#OrderShipped"
   process:hasOutput
    process:ConditionalOutput rdf:ID="CongoOutOfStockOutput"
      process:coCondition rdf:resource="#OutOfStock"
       process:parameterType rdf:resource="#OutOfStock"

process:hasInput
    process:Input rdf:ID="ExpressCongobookISBN"
     process:parameterType rdf:resource="&ISBN"
   process:hasInput
    process:Input rdf:ID="CongoSignInInfo"
     process:parameterType rdf:resource="#SignInData"
   (...)
 process:hasPrecondition rdf:resource="#hasAcctID"
   process:hasPrecondition rdf:resource="#ValidCreditCard"
(...)

   rdf:RDF (...) xmlns = "&BankAndFinance;#"
   owl:Ontology rdf:about=""
     (...)
   owl:Class rdf:ID="CreditCard"
     rdfs:subClassOf
       owl:Restriction
         owl:onProperty rdf:resource="#cardNumber"
         owl:cardinality rdf:datatype="&xsd;#integer"
           1
     (...)
   owl:DatatypeProperty rdf:ID="cardNumber"
     rdfs:domain rdf:resource="#CreditCard"
     rdfs:range rdf:resource="&xsd;#string"
     (...)
```

Fig. 4. Schematic view of transformations between UML and OWL-S

cantly extend the scope of the paper and thus is not handled by our transformations. The inputs, outputs, pre- and postconditions, and effects are generated at two places in the OWL-S document: the Process section and the Profile section. The transformation generates the elements in the Process part first. Then, these elements are basically duplicated for the Profile part. In fact, the referring elements found in the Profile section can be seen as a summary.

Reused ontologies are modeled in UML as separate packages with a URI as tagged value to identify the ontology. All such ontologies result in an import statement in the produced OWL-S document. Then the ontology concepts belonging to an imported ontology can be used at the adequate places (such as parameterType) by combining a short name of the imported ontology and the full name of the ontology concept. For



OWL-S	Standard UML 2.0 activity model elements
Sequence	Sequence
Unordered	-
Split	DecisionNode or Fork
Split+Join	Fork + Join
Choice	DecisionNode
If-Then-Else	DecisionNode
Repeat-Until	LoopNode
Repeat-While	LoopNode
Iterate	LoopNode

Fig. 5. FullCongoBuy service manually reverse engineered into UML

each new ontology concept a new owl:class is created. We do not explain further how to facilitate this part of the transformation, which is outlined with the box labelled 8, as this is covered by Duric's work about representing OWL ontologies in UML [1].

The transformation can be extended to also handle conversations of Web services. A conversation occurs when several operations have to be called in a specific order before the service is completed. The OWL-S CompositeProcess element describes Web services that are to be called in a conversational manner. In OWL-S, the conversation of a Web service is described by a CompositeProcess element. The CompositeProcess represents the conversation model by using control flow constructs to connect other contained OWL-S Processes. An OWL-S CompositeProcess is a Web service composition which corresponds to a sub activity in UML. All the available control flow elements in OWL-S are shown in the table of Figure 5. The table shows corresponding UML elements for each of the OWL-S control flow constructions, except for the unordered element which have no natural corresponding element in UML. The unordered element contains a collection of Web services that shall be called in a sequence, but the order may be arbitrarily. A new stereotype called Unordered based on a sub activity could be introduced where the content is restricted to a set of unconnected activities. The single OWL-S Split+Join element corresponds to two separate elements in UML, a Fork followed by a join. Repeat-Until, Repeat-While and Iterate in OWL-S can all be mapped to UML LoopNode. However the LoopNode has no defined graphical notation, which then remains to be solved in order to have a visual representation. The rest of the mappings in the table are trivial.

The Congo example consists of two parts: ExpressCongoBuy and FullCongoBuy. The ExpressCongoBuy is a single Web service operation which we have already shown a representation of in our UML profile. The right part of Figure 5 shows how the conversational OWL-S FullCongoBuy example can be imported into a UML activity model by using the transformations defined in the table on the left side. Note that this is a simplified UML model which contains only the control flow and the individual service names. First the LocateBook service is called. The resulting output indicates if the book is in stock or not. The UML decisionNode control flow construct is introduced as a conditional OR-split that will finish the interaction if the book is not in stock. Otherwise, the composite service BuyBook is called, which involves a control flow graph of its own. A sequence of service calls is performed to put the book in cart, get the sign in alternatives, provide payment details, provide delivery information, and finally the complete Web service conversation ends.

5 Correspondence Between WSML, OWL-S and Our UML Profile

This Section explains the main parts of WSML and how this relates to OWL-S and our UML profile. WSML consists of four main parts: Ontologies, Goals, Mediators and Web services. The Ontologies part in WSML corresponds to OWL ontologies in an OWL-S document or to the UOP part of the UML profile. Goals represents abstract services for which one desires to find realizing Web services that can satisfy the requirements. There is no support for goals in OWL-S or in our specified UML profile. However, a new stereotype called Goal could be introduced which shares all the properties of the WebService concept except the four WSDL tagged values. Mediators deal with transformations and relationships that can be categorized as four kinds: OO-, WW-, GG- and WG-mediators. The OO-Mediator is the translation from one ontology to another. The WW-Mediator handles interoperability between Web services. The GG-Mediator expresses relationships between different goals. The WG-Mediator matches a goal with a realizing Web service. There is no equivalent to the mediators in OWL-S and its functionality is outside the scope of our UML profile.

The Web services part of WSML corresponds to our semantic Web service profile in UML and the OWL-S focus. The table on the left side of Figure 6 shows the correspondence between elements in the UML profile and the elements of WSML. The right side of the figure shows how the previously described ExpressCongoBuy can be expressed in WSML. We have simplified the WSML example by omitting the full definitions of the logical expressions for pre-condition, post-condition and effect. Many of the concepts have trivial mappings, but the definition of input and ouput parameters in the interface is quite different in WSML. The concepts to be used in the Web service interface are imported from ontologies and specialized by the WSML *subConceptOf* keyword. Namespaces are used to separate the imported super type from the new sub type. The sub type defines if the concept is used as an input or output parameter. The

UML Profile	WSML
WebService	webservice
input	imported concepts with mode hasValue *wsml#in*
output	imported concepts with mode hasValue *wsml#out*
pre-condition	precondition
post-condition	postcondition
effect	effect
Ontology (UOP)	ontology
OntClass (UOP)	concept

```
webservice http://.../ExpressCongoBuy
    nonFunctionalProperties
        dc:title hasValue "ExpressCongoBuy"
        dc:creator hasValue "DERI International"
        dc:description hasValue "This is an express ..."
        ...
    endNonFunctionalProperties

    importedOntologies {<<http:://.../Shipment>>,
        <<http:://.../BankAndFinance>>,
        <<http:://.../PrintedMaterial>>}

    capability
        precondition <hasAcctID> + <ValidCreditCard>
        postcondition <bookExistsForISBN> + <bookOutOfStock>
        effect <Shipment of book>

    concept bookISBN subConceptOf pm-namespace#bookISBN
        nonFunctionalProperties
            wsml#mode hasValue wsml#in
            wsml#grounding hasValue PrintedMaterial#ISBN
        endNonFunctionalProperties
    concept signInInfo ...
    concept creditCardNumber ...
    concept creditCardType ...
    concept creditCardExpirationDate ...
    concept output1 ...
        nonFunctionalProperties
            wsml#mode hasValue wsml#out
            wsml#grounding hasValue OrderShippedAcknowledgement
        endNonFunctionalProperties
    concept output2 ...
```

Fig. 6. Left: Mapping our UML profile to WSML. Right: ExpressCongoBuy in WSML

bookISBN parameter is shown with a complete WSML definition which illustrates the use of *subConceptOf*.

6 Evaluation

This Section evaluates the UML profile for semantic Web services against our requirements. The requirements which we have mentioned in Section 1 are:

– expressive enough for semantic Web services,
– independence of the lexical semantic Web service language, and
– readability for human readers.

Expressiveness. We have implemented a reverse transformation from OWL-S to UML and a forward transformation from UML to OWL-S. Both are implemented in the UML Model Transformation Tool (UMT) [7]. UMT is based on the XML Metadata Interchange Format (XMI) [10] which allows the realization of transformations that are UML tool-independent. The transformations between UML and OWL-S show to a large extent that our UML profile is expressive enough to capture and generate the needed semantic information of OWL-S. There are however three parts that are not implemented and that needs further extensions to make the profile fully expressive: metadata (such as contact name, address etc.), logic expressions and control flow. The support for organizational metadata about the service can be regarded as trivial. We have also suggested how the available control flow constructs can be handled, while logic expressions need more investigation to be covered properly.

The transformation from OWL-S to UML have been verified by testing on the Congo and Bravo OWL-S reference examples. The parser of a previously developed OWL-S matchmaking implementation [9] has successfully processed the OWL-S output of the UML to OWL-S transformation. However, we regard this test as a proof-of-work but not as a formal verification.

Independence. The independence of the lexical semantic Web service language cannot be fully claimed as we have not implemented any transformation between UML and the competitor WSML. However, we have explained how to fully support OWL-S and how to cover the relevant concepts of WSML in conjunction with our UML profile. Furthermore, the constructs we have proposed in our UML profile are not specifically designed to match one particular semantic Web service language.

Readability. The approach taken in this paper of using annotated UML activity models may be compared with a different approach of using UML interfaces with operations. UML Interfaces have the strength that they better define the separation between input and output parameters for which we needed to introduce stereotypes. We also need to be careful in the way the activity parameters are laid onto the activity depending on the incoming and outgoing flow when it is integrated into a composition model. If the input parameters are placed on the left side, then the incoming flow should also hit the left side boundary of the activity in order not to confuse a human interpreter.

A disadvantage of using UML interfaces is that comments, pre- and postconditions belonging to an input or output are not easily visualized in the diagram of an interface element, while this is clearly visualized in our approach. The same argument goes for individual operations within an interface. These are handled properly in our approach since all operations are mapped to separate activities where comments, preconditions, postconditions, effects, category description etc. can be attached.

7 Related Work

We have already covered the two semantic Web service languages, OWL-S and WSML, that have the largest attention at the moment. Two other semantic Web service languages have also been proposed recently. WSDL-S [13] extends WSDL 2.0 with semantic descriptions. This language uses OWL types instead of the XML Schema syntactical data types to define the parameter types in operations. There are no extensions defined for expressing logical constraints such as pre-conditions, post-conditions and effects. Hakimpour et al. [8] shows how The Operational Conceptual Modeling Language (OCML) is used by the IRS-III system to enable automatic service composition. They discuss differences and similarities between OCML, OWL-S and WSML. None of these two additional semantic Web service languages seem to introduce concepts that are not covered by our UML profile.

Duric explains in his work [1] how OWL concepts can be transformed into the UML Ontology Profile. This work is adopted as part of our transformation strategy from OWL-S to UML, and the domain ontology modeling part of this paper does not contain any new aspects. Elenius et al. [3] present the OWL-S editor as a graphical tool with the ability to both import and export OWL-S documents. Their approach is OWL-S-dependent, while our approach delivers a graphical language and transformations that can be reused towards different semantic Web languages, where OWL-S is just one candidate. Scicluna et al. [14] have a similar approach to ours by using UML 2.0 activity models to represent semantic Web services with a generation to OWL-S. Their approach is also tailored for OWL-S. Gomez-Perez et al. [6] have identified the same requirement as we have to deliver a semantic Web language-independent graphical language. To fulfill this need they propose the proprietary ODE SWS graphical language for modeling tasks that can be associated with inputs, outputs, preconditions and postconditions. In addition to their work we have defined transformation rules between OWL-S and UML. The inputs and outputs in ODE SWS are modeled as separate data objects on the outside of the task quite similar to UML 1.5 Activity models. In our approach we have used the more compact pin-notation of UML 2.0 Activity models that attaches a small symbol to the boundary, which makes the interface easier to read.

The METEOR-S tool presented by Rajasekaran et al. [13] is an approach where a user can annotate operations and its parameters with pre-conditions, post-conditions and semantic types in a tree view browser. From this tree view both WSDL-S and OWL-S generations have been implemented which shows a semantic language-independent approach. However, importing existing semantic Web service documents is not possible in this tool and compositions with control flow is not covered. Peer [12] proposes a lexical language to define imperative constructions with complex goals that are used to automatically generate executable Web service compositions. The executable Web service compositions can be displayed with a graphical interface showing control flow and semantic annotations, but importing and exporting semantic Web service documents is not supported.

8 Conclusions

The contributions of this paper are a UML profile for semantic Web service composition and transformations both ways between UML and OWL-S. By importing semantic OWL-S descriptions of existing Web services into UML diagrams, we show that UML can be used as a common integration platform. The ability to generate Web service composition documents with semantic descriptions from a graphical model represents a valuable gain to the service developers, who otherwise have to write a lot of low-level XML code.

Fully defined transformations between UML and WSML is also desired so that our UML profile also can be used with the other leading semantic Web service specification. Another future improvement is to enhance the transformations between UML and OWL-S by also handling the logical expressions to cover the pre- and postconditions and the effects. This could be achieved by defining and implementing transformations between the logical languages used by OWL-S and WSML and the Object Constraint Language in UML. We see this as the next step towards providing a user-friendly environment to interpret and define the logical expressions.

Acknowledgements. The work of SINTEF is partially funded by the European IST-FP6-004559 project SODIUM (Service Oriented Development In a Unified framework). We would also like to thank Martin Hepp from DERI for his valuable input.

References

1. Dragan Djuric. MDA-based Ontology Infrastructure. *Computer Science Information Systems (ComSIS)*, 1(1):91–116, February 2004.
2. Satish Tatte (Editor). Business Process Execution Language for Web Services Version 1.1. Technical report, BEA Systems, IBM Corp., Microsoft Corp., http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, February 2005.
3. Daniel Elenius, Grit Denker, David Martin, Fred Gilham, John Khouri, Shahin Sadaati, and Rukman Senanayake. The OWL-S Editor - A Development Tool for Semantic Web Services. In *2nd European Semantic Web Conference (ESWC 2005)*, Heraklion, Crete, Greece (accepted for publication), May 2005.
4. David L. Martin et al. Bringing Semantics to Web Services: The OWL-S Approach. In *Semantic Web Services and Web Process Composition, First International Workshop, SWSWPC 2004, Revised Selected Papers*, Volume 3387 of *Lecture Notes in Computer Science*, San Diego, California, USA, July 2004. Springer.
5. Ian Horrocks et al. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Technical Report, http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/, May 2004.
6. Asunción Gómez-Pérez, Rafael González-Cabero, and Manuel Lama. ODE SWS: A Framework for Designing and Composing Semantic Web Services. *IEEE Intelligent Systems*, 19(4):24–31, 2004.

7. Roy Grønmo and Jon Oldevik. An Empirical Study of the UML Model Transformation Tool (UMT). In *The First International Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA)*, Geneva, Switzerland, February 2005.
8. Farshad Hakimpour, John Domingue, Enrico Motta, Liliana Cabral, and Yuangui Lei. Integration of OWL-S into IRS-III. In *First AKT Workshop on Semantic Web Services, AKT-SWS04*, Athens, Greece, December 2004.
9. Michael C. Jaeger, Gregor Rojec-Goldmann, Gero Mühl, Christoph Liebetruth, and Kurt Geihs. Ranked Matching for Service Descriptions using OWL-S. In *Kommunikation in verteilten Systemen (KiVS 2005), Informatik Aktuell*, Kaiserslautern, Germany, February 2005.
10. Object Management Group (OMG). XML Metadata Interchange (XMI) Specification v1.2, OMG Document: formal/02-01-01. Technical report, January 2002.
11. Object Management Group (OMG). UML 2.0 Superstructure Specification, OMG Adopted Specification ptc/03-08-02. Technical Report, August 2003.
12. Joachim Peer. A PDDL Based Tool for Automatic Web Service Composition. In *Proceedings of the Second International Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR)*, St. Malo, France, September 2004.
13. Preeda Rajasekaran, John A. Miller, Kunal Verma, and Amit P. Sheth. Enhancing Web Services Description and Discovery to Facilitate Composition. In *Semantic Web Services and Web Process Composition, First International Workshop, SWSWPC 2004, Revised Selected Papers*, volume 3387 of *Lecture Notes in Computer Science*, San Diego, California, USA, July 2004.
14. James Scicluna, Charlie Abela, and Matthew Montebello. Visual Modelling of OWL-S Services. In *Proceedings of the IADIS International Conference WWW/Internet*, Madrid Spain, October 2004.
15. World Wide Web Consortium (W3C). SOAP Version 1.2 Part 0: Primer. Technical Report, http://www.w3.org/TR/soap12-part0/, June 2003.
16. World Wide Web Consortium (W3C). Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. Technical Report, http://www.w3.org/TR/wsdl20, August 2004.
17. WSMO working group. D16.1v0.2 The Web Service Modeling Language WSML, WSML Final Draft. Technical Report, March 2005.

A Graphical Specification of Model Transformations with Triple Graph Grammars

Lars Grunske[1], Leif Geiger[2], and Michael Lawley[3]

[1] School of Information Technology and Electrical Engineering,
University of Queensland, Brisbane, QLD 4072, Room 72-458 IT Building
grunske@itee.uq.edu.au
http://www.itee.uq.edu.au/

[2] University of Kassel, Software Engineering Research Group,
Department of Computer Science and Electrical Engineering,
Wilhelmshöher Allee 73, 34121 Kassel, Germany
leif.geiger@uni-kassel.de
http://www.se.eecs.uni-kassel.de/se/

[3] CRC for Enterprise Distributed Systems Technology (DSTC)*,
University of Queensland, Brisbane, QLD 4072, Australia
michael@lawley.id.au
http://www.dstc.edu.au/

Abstract. Models and model transformations are the core concepts of OMG's MDA^{TM} approach. Within this approach, most models are derived from the MOF and have a graph-based nature. In contrast, most of the current model transformations are specified textually. To enable a graphical specification of model transformation rules, this paper proposes to use triple graph grammars as declarative specification formalism. These triple graph grammars can be specified within the FUJABA tool and we argue that these rules can be more easily specified and they become more understandable and maintainable. To show the practicability of our approach, we present how to generate Tefkat rules from triple graph grammar rules, which helps to integrate triple graph grammars with a state of a art model transformation tool and shows the expressiveness of the concept.

1 Introduction

Model Driven Engineering (MDE) is a software engineering principle that promotes the use of models and transformations as primary development artifacts. To practically apply this principle, the Object Management Group (OMG) has proposed the MDA$^{\text{TM}}$ [1] as a set of standards for integrating MDE tools. The MDA$^{\text{TM}}$ approach separates the specification of systems from the implementation of these systems. For this reason two basic model types are introduced,

* The work reported in this paper has been funded in part by the Co-operative Research Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme.

Platform Independent Models (PIM's) and Platform Specific Models (PSM's) using specific implementation platforms. PIM's can be specified in an abstract style without thinking about platform specific details. If all necessary PIM's are specified, they should be automatically mapped to a platform specific model by adding the platform specific details. To allow this mapping, model transformations are necessary.

The need for standardization of model transformations as well as the generation of views and the definition of queries lead to the MOF 2.0 Query/ Views/ Transformations Request for Proposals (RFP) [2] from the OMG. For this RFP the OMG initially received eight proposals of varying degrees of completeness which are reviewed and assessed in [3]. A final revised, merged submission supported by all original submitters is expected to be voted for adoption at the June 2005 OMG meeting in Boston. This submission supports several flavours of transformation specification allowing for both declarative and procedural specifications.

One important aspect we noticed when reviewing the revised submissions for the RFP is that most transformation languages are specified textually. This conflicts with the graph-based nature of most of the current MOF 2.0 models (e.g. UML 2.0 models). A graph-based transformation language would be more appropriate for specifying and applying model transformations. Having made this observation, we propose to use graph transformations rules (specifically triple graph transformation rules) and graph transformation systems as an extension to the current model transformation languages. These graph transformation rules are a straightforward extension of string or term rewriting rules, which were introduced in the seventies [4] and are currently applied in various domains [5] to transform or rewrite graph-based structures. However, normal graph transformation rules and systems are only suitable to operate on one particular graph. Thus, they are only suited to describe intra-model transformation, as they are needed to specify quality-improving refactorings [6,7,8]. To operate on two different graphs with two different graph schemata an extension called triple graph grammars [9] is suitable. These triple graph grammars and their rules are the theoretical foundation of this paper and we want to show their suitability for describing complex model-to-model transformations. Especially, we argue that triple graph grammars provide the following benefits, which are useful for the specification and application of model-to-model transformations [10]:

– Triple graph grammars allow incremental change propagations between two models A and B, if a model transformation system with triple graph grammar rules is applied once to transform a model A to a model B and the correspondence graph is generated. This means you can create tools that update a model if the other has been changed. [11,9] This change propagation is also bidirectional and is especially important for iterative software engineering processes where a model evolves continuously.
– Triple graph grammars can be used to check the consistency between two models.

– Triple graph grammars can be applied to all graph-based data structures and models, not only to tree-based ones. This also includes hierarchical graph-based data structures [12,13,14], which we think is imported to model transformations between two MOF 2.0-compliant models.

The rest of the paper is structured as follows: Section 2 introduces the basic concepts of graph-based structures and graph transformations in general. Thereafter, Section 3 reviews the concepts of triple graph grammars and shows how triple graph grammar rules can be used to specify model-to-model transformations. In Section 4, the practical applicability of triple graph grammars is presented with the well-known example of the transformation from an object-oriented class diagram model into a relational database model. Section 5 presents an implementation of triple graph grammars within the FUJABA tool and describes how FUJABA could be used to generate textually specified model transformation rules (e.g. Tefkat rules). Before concluding, Section 6 discusses the limitation of triple graph grammars and sets up directions for future work to extend the current model-to-model transformation languages successfully.

2 Preliminaries

This section introduces the basic concepts of graph-based structures and the fundamental graph transformation theory in an informal and intuitive way. In addition, this section presents an overview about useful graph transformation techniques and extensions that are needed to specify model transformations within the MDA$^{\text{TM}}$ approach.

2.1 Directed Typed Graphs and Graph Morphisms

We choose directed typed graphs as the basic structure, because they are well suited to specifying different types of models, especially MOF-based models [15]. These directed typed graphs contain nodes and edges that are instances of node and edge types. The instance relation between the nodes and edges and their types is similar to the relation between objects and classes in object-oriented software engineering. Due to this, a node or edge type can contain a set of application specific attributes and operations. To model the graph-based structure each edge is associated with a source and a target node. Formally, a typed graph can be defined as follows:

Definition 1. (Directed Typed Graphs) Let L_V be a set of node types and L_E be a set of edge types; then a directed typed graph G from the possible set of graphs $\mathcal{G}$ over L_V and L_E is characterized by the tuple $\langle V, E, \mathit{source}, \mathit{target}, \mathit{type}\rangle$, with two finite sets V and E of nodes (or vertices) and edges, a function *type* composed of the two functions $\mathit{type}_V : V \rightarrow L_V$ and $\mathit{type}_E : E \rightarrow L_E$ which assigns a type to each edge and node and two functions $\mathit{source} : E \rightarrow V$ and $\mathit{target} : E \rightarrow V$ that assign to each edge a source and a target node.

Another preliminary for the definition of graph transformation systems are graph morphisms. These graph morphisms are structure and type-preserving mappings between two graphs that can be defined as follows:

Definition 2. (Graph Morphism) Let $G = \langle V, E, \mathit{source}, \mathit{target}, \mathit{type}\rangle$ and $G' = \langle V', E', \mathit{source}', \mathit{target}', \mathit{type}'\rangle$ be two graphs; then a graph morphism $m : G \to G'$ consists of a pair of mappings $\langle m_V, m_E\rangle$, with $m_V : V \to V'$ and $m_E : E \to E'$, which satisfy the following conditions (type and structure-preserving):

– $\forall\, e \in E : \mathit{type}'(m_E(e)) = \mathit{type}(e)$
– $\forall\, v \in V : \mathit{type}'(m_V(v)) = \mathit{type}(v)$
– $\forall\, e \in E : \mathit{source}'(m_E(e)) = m_V(\mathit{source}(e))$
– $\forall\, e \in E : \mathit{target}'(m_E(e)) = m_V(\mathit{target}(e))$

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

Graph Variants. Besides the introduced directed typed graphs, several other variants and extensions gain attention in the graph transformation community. One basic variant uses undirected edges. These undirected edges can be modeled in a directed graph with two contrary edges for each undirected edge. Another variant are hypergraphs[16], where each (hyper) edge is associated with a sequence of source and target node. That means, these edges can have an arbitrary number of source and target nodes. For the construction of hierarchical models, hierarchical graphs are important. These hierarchical graphs model the hierarchical structure either by (hyper)edge [12] or node refinement [17].

2.2 Graph Transformation and Graph Transformation Systems

Basic Principles. Graph transformation systems make use of graph rewriting techniques to manipulate graphs. A graph transformation system is defined with a set of graph production rules, where a production rule consists of a left-hand side (LHS) graph and a right-hand side (RHS) graph. Such rules are the graph equivalent of term rewriting rules, i.e., intuitively, if the LHS graph is matched in the source graph, it is replaced by the RHS graph. Intuitively, a graph transformation rule can be defined as follows:

Definition 3. (Graph Transformation Rule) A graph transformation rule $p = \langle G_{LHS}, G_I, G_{RHS}, m_l, m_r\rangle$ consists of three directed typed graphs G_{LHS}, G_I and G_{RHS}, which are called left-hand side graph, interface graph and right-hand side graph. The interface graph G_I is just an auxiliary graph. The morphisms $m_l : G_I \to G_{LHS}$ and $m_r : G_I \to G_{RHS}$ are used to describe the correspondence between these graphs and map the elements of the interface graph to either the left-hand side or the right-hand side graph.

For the application of a graph transformation rule to an application graph G_{APP} the following simplified algorithm can be used:

1. Identify the left-hand side G_{LHS} within the application graph G_{APP}. For this, it is necessary to find a total graph morphism $m : G_{LHS} \rightarrow G_{APP}$ that matches the left-hand side G_{LHS} in the application graph G_{APP}.
2. Delete all corresponding graph elements, w.r.t. m, in the application graph G_{APP} that are part of the left-hand side G_{LHS} and are not part of the interface graph G_I.
3. Create a graph element in the application graph G_{APP} for each graph element that is part of the right-hand side G_{RHS} and is not part of the interface graph G_I. Connect or glue these added graph elements to the rest of the application graph G_{APP}.

For a formal description of the rule application formalisms, we refer to [18,19], where the formal foundations of the single pushout (SPO) and double pushout (DPO) approach are reviewed. Currently these approaches have the most impact in the graph transformation community [5,15,8].

Application Conditions. In complex graph transformation systems it is often necessary to restrict the application of single rules. Therefore, in [20] the concept of positive and negative application conditions (PACs and NACs) is introduced. These application conditions are formally graphs that define a required context (PACs; e.g. the presence of nodes or edges) or a forbidden context (NACs; e.g the absence of nodes or edges). The fulfillment of these application conditions must be checked before the rule is applied. Consequently, the algorithm in the previous Section must be extended by adding another step after the first one, which checks the application conditions. In this paper, we use crossed out nodes to visualize nodes belonging to the negative application condition.

Specification of Graph Transformation Rules. In traditional approaches for specification of graph transformation rules the right hand side and the left hand side of a rule are drawn separately [19,18]. Throughout this paper the approach of the Fujaba Tool [21], that combines both sides, is used. Fujaba uses UML collaboration diagrams to model graph transformations. Nodes become objects and edges become links between objects. Objects and links that have no stereotypes appear on both sides of the graph transformation rule. Objects and links marked with the ≪*destroy*≫ stereotype appear only on the left hand side, i.e. they are deleted. The stereotype ≪*create*≫ marks elements only used on the right hand side, i.e. such elements are created. Fujaba uses programmed graph transformation rules. This means a control structure can be specified that manages the order of the execution of transformation rules. Such control structure is modeled using UML activity diagrams. The transformation rules are then embedded into the activities. Fujaba makes use of typed graphs. In the graph transformations, the type is specified after the object's name, and separated by a colon. More elaborate elements of graph transformations like negative application conditions, multi objects and non-injective matching are also supported by the Fujaba tool. All of these features will be needed for the effective specification of model transformation rules.

Figure 1 shows a graph transformation rule in Fujaba. The rule consists only of one transformation that deletes every column in a table that has the same name as another column. To achieve this, the transformation tries to match an object *c1* of type *Column* which has a *col* link to a *Table* object *t*. This object must itself have a *col* link to another object of class *Column*. If this *column* has the same attribute value for its attribute *name* as the object *c1*, the matching can be applied. If a matching is found, the column *c2* and its *col* link will be destroyed. Note, that the activity has a doubled border. Such an activity, a so-called for-each activity, is applied as long as a matching is found. Thus, the rule deletes every duplicated column in every table.

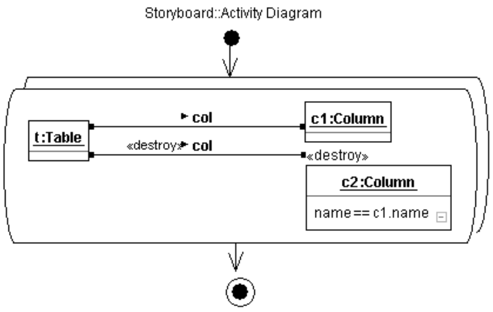


Fig. 1. Graph transformation "Fujaba-style"

3 Triple Graph Grammars

In this section we introduce the concept of triple graph grammars (TGG) and describe their suitability to extend current model transformation systems. Thereafter, we show how triple graph grammars can be specified within the Fujaba tool and how forward and backward transformations can be derived from these triple graph grammar rules.

3.1 Introduction

Triple graph grammars are a straightforward extension of pair grammars and pair grammar rules that were introduced by Pratt [17] in the early seventies. These pair grammars are used to specify graph-to-string translations. By this means, a pair grammar rule rewrites two models: a source graph and a target string. Thus, it contains a pair of production rules (a graph and a string production rule), which modify simultaneously the two participating models. Because of this, pair grammars are well suited to specify transformations between graphs and strings. If the string production rule is substituted by a graph production rule, these pair grammars can be also used for graph-to-graph translations.

Triple graph grammars, as introduced in the early nineties [22] are used for graph-to-graph translations and data integration. Each triple graph grammar rule contains three graph productions; one operates on a source graph, one on the target graph and one on a correspondence graph. The correspondence graph describes a graph-to-graph mapping that relates elements of the source graph to elements of the target graph. Based on this mapping, incremental change propagations, that update the target graph if an element in the source graph is changed, are possible. Formally, triple graph grammar rules can be defined as follows [22]:

Definition 4. (Triple Graph Transformation Rule) A triple graph transformation rule $tgg = \langle p_{left}, p_{right}, p_{map} \rangle$ consists of three graph transformation rules p_{left}, p_{right} and p_{map} where p_{left} transforms the source model, p_{right} transforms the target model and p_{map} transforms a relation model that maps source to target elements. All three graph production are applied simultaneously.

3.2 Specification

To specify a complex triple graph grammar rule all three graph grammar rules should be specified in one rule diagram. For the specification of each single rule we use the Fujaba-style and separate the three rules within the rule diagram. Due to this separation a user can identify to which side the element belongs.

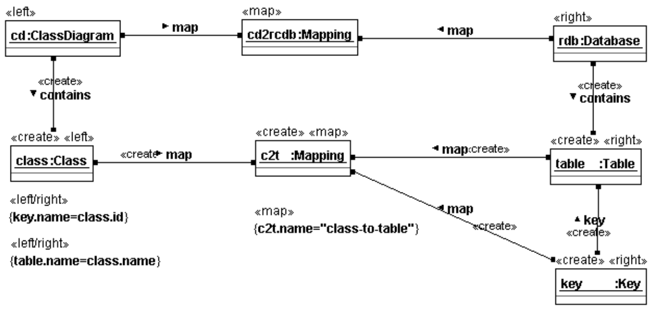


Fig. 2. Example of a TGG rule class-to-table in FUJABA

Figure 2 shows a transformation rule that contains seven objects; two source model objects, three target model objects and two correspondence model objects. The objects from the source model are drawn left, the objects of the target model are drawn right and the objects of the correspondence model are drawn in the middle of the rule diagram. Additionally, they are marked with the stereotypes ≪*left*≫, ≪*map*≫ or ≪*right*≫. The rule shown in the figure demonstrates a mapping between classes in a class diagram and tables in a relational database. The precondition of this rule is drawn in the top of the diagram. This means, to apply this rule there must already exist a class diagram which is mapped via a *Mapping* node to a relational database. The elements which have to be related are drawn green with a ≪*create*≫ stereotype. This means, an object of the type *Class* is mapped to an object of the type *Table* which has a link to a key object and vice versa. Attribute conditions are modeled as constraints. For example, the *id* of a class is stored as *name* of the table's key.

A triple graph grammar rule enable the generation of three transformation rules [9]: the forward rule, the reverse rule and a relation rule that checks the consistency of both models. The forward rule is created by removing the ≪*create*≫ stereotype from all elements which belongs to the source (≪*left*≫) model. The reverse rule is created by removing the ≪*create*≫ stereotype from all elements which belongs to the target (≪*right*≫) model. The last rule, the relation rule, is derived by removing the ≪*create*≫ stereotype from all elements that does not belong to the correspondence (≪*map*≫) model.

Figure 3 shows the derived forward rule. If a mapping from a class diagram to a relational database exists and if the class diagram contains a class, a new table and a new key are created and its attributes are set accordingly to the TGG rule. These newly created objects are marked as being mapped to the class using a new mapping node.

Figure 4 shows the reverse rule, which will create a class for every table in the relational model. The matching and creation of objects is done in the same manner as is done for the forward rule.

The last rule created is the relation rule shown in Figure 5. This rule needs a class diagram and a database and tries to relate them. This will result in a consistency check between these two models. Therefore, the only objects created in this rule are the mapping nodes.

The rule in Figure 5 searches for a class diagram that has already been mapped to a relational database. If the class diagram contains a class which can be related to a table as specified in the rule, a new mapping is created. If

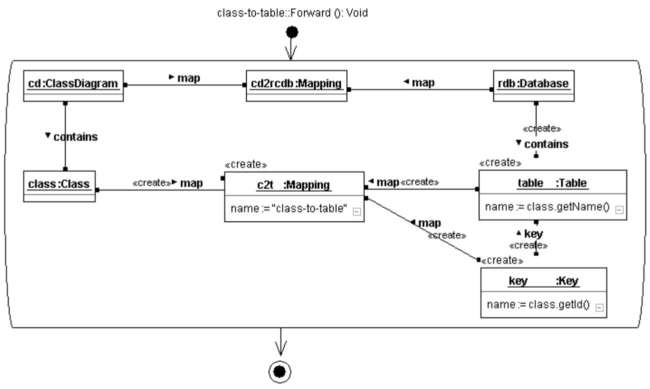


Fig. 3. Forward rule derived from the TGG-rule class-to-table

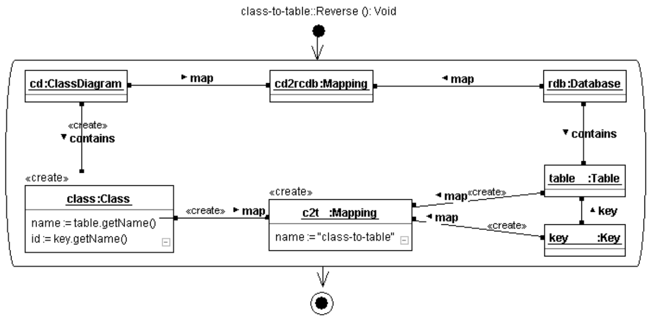


Fig. 4. Reverse rule derived from the TGG-rule class-to-table

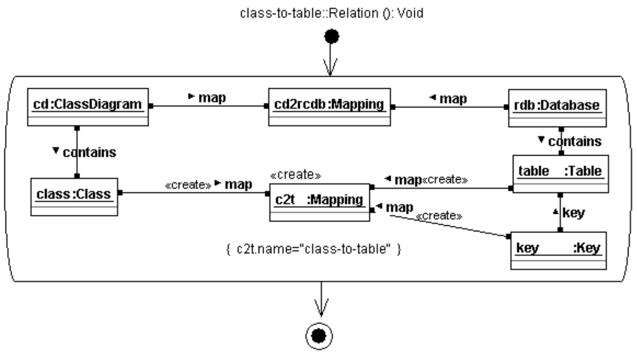


Fig. 5. Relation rule derived from the TGG-rule class-to-table

a mapping can be created for all classes and for all tables the two models are consistent.

4 Example

To show the practical applicability of triple graph grammars in the context of model transformations we use the well-known example of the transformation from an object-oriented class model to a relational database model. This transformation is required if an application needs to store a set of objects persistently in a database. A text-based realisation of this example can be found in several QVT-proposals [23,24]. A graphical specification of the transformation rules of this example can be found in [25].

The basic meta-models, the object-oriented class model and the relational database model, for this transformation are presented in Figure 6. To keep the transformations simple the object-oriented model is cut down in the following aspects:

- A class can contain only attributes and no methods, because methods don't need to be stored persistently.
- Only 1:1 and 1:n associations are considered. These associations are represented by attributes that have classes as types. For a modelling of m:n associations a new meta class *Association* need to be introduced, that has a source and a target association to the meta-class *Class*.
- In some examples, the meta-class *Classifier* or *Class* has a Boolean attribute *ispersistent* that is used to mark all objects that need to be stored in the database. For simplicity in our transformation we transform the complete object-oriented model into a relational model.

The effect of these simplifications will be described and discussed in the following, when we describe the transformations in detail.

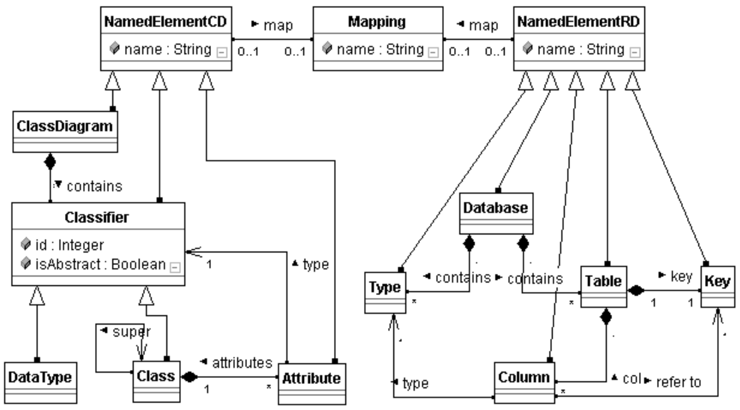


Fig. 6. Meta-models including the mapping relation

To transform an instance of the object-oriented meta-model into the relational model the following natural language rules(requirements/laws) are used:

– *Classes* correspond to *Tables* that have a unique *Key*. This *Key* is identical to the *id* of the *Class*.
– *Types* in the relational model correspond to simple *Datatypes* in the object-oriented model.
– *Attributes* are stored in *Columns*, where each *Column* is owned by the *Table* of the corresponding *Class*.

To implement the transformations and consistency checks between the object-oriented class model and the relational database model a set of triple graph grammar rules must be created for each law. The triple graph grammar rule for the first law, the mapping from classes to tables, has already been discussed in Section 3.2. The rule for the second law is very similar. This rule relates each *Datatype* in the object-oriented model and each *Type* in the relational model the same way as it is done for the first law.

To store the attributes in columns and vice versa, as requested in the third law, we need to distinguish between attributes of a simple type and attributes that are classes. Due to this reason, we need to specify two different TGG-rules (cp. Figure 7). To get an impression what these rules do, we now have a look at the forward rules, which can be generated from the two TGG-rules. The first rule searches for all *Attributes* of a *Class* that are typed by a DataType. It then creates a *Column* for each *Attribute* and assigns the *Column* to the *Table* of the *Class* and to the *Type* of the corresponding *Attribute*. The second rule tries to match all *Attributes* of a *Class* that refer to another *Class*. If this rule finds such a match, it creates a new *Column* and assigns it to the *Table* of the *Class* to which the *Attribute* belongs. To set the correct *Type* of the *Attribute* the *Table*

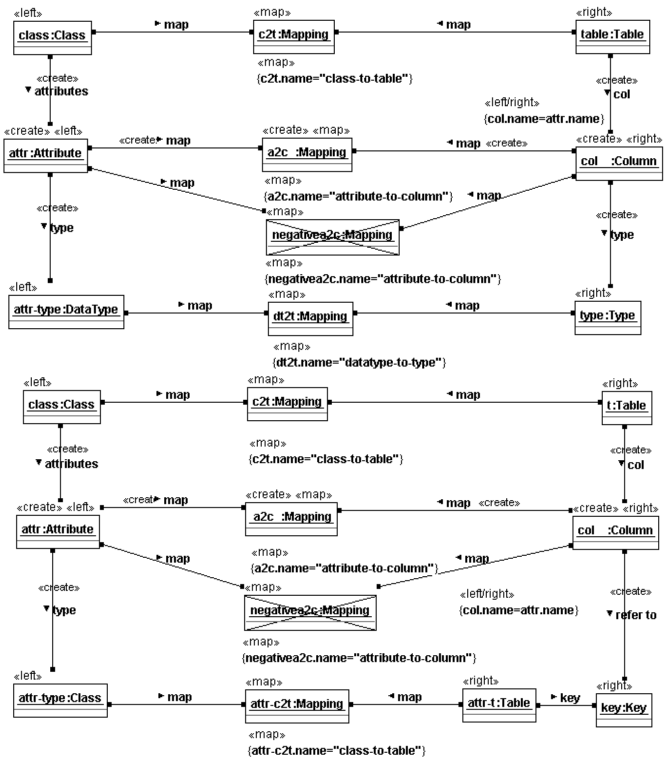


Fig. 7. TGG-rules for attributes and columns

of the *Class* is identified with the mapping relation and the association *refer to* is set to the *Key* of this *Table*.[4,5]

5 Implementation

For specifying triple graph grammar rules we use FUJABA's [21] TGG Plug-in, which provides all necessary triple graph grammar concepts for specifying model transformations. Additional, the TGG-Plug-in is open source and easily extendable.

[4] Note that before these rules can be applied it is necessary to apply the rules that transform *Classes* and *Datatypes*. If these rules are not previously applied, the mapping relations are missing and the LHS of the TGG-rule can't be matched. This leads to an implicit specification of the ordering of the rules.

[5] With the described transformations of attributes that are typed by classes, all 1:n and 1:1 associations can be transformed. To transform the m:n associations an additional rule is necessary. This rule creates a new table for all m:n associations and stores the links to the corresponding classes as foreign keys in this table.

```
RULE class-to-tableForward(class, cd, rdb, key, table)
 FORALL Class class, ClassDiagram cd, Database rdb
  WHERE cd2rcdb LINKS cd=cd, db=rdb
        AND cd.contains=class
   MAKE Key key, Table table
    SET key.key=table,
        rdb.contains=table,
        key.name=class.id,
        table.name=class.name
LINKING c2t
   WITH table=table, class=class, key=key;
```

Fig. 8. Forward Rule in Tefkat

To execute model transformations, that are specified as TGG rules, two options are provided by the FUJABA tool. As a first option, story diagrams could be generated from TGG rules. These story diagrams can be used to generate Java code, which enables a so called in-memory model transformation [26].

Based on this approach we have implemented a second alternative, which generates rules that can be used in the Tefkat tool [27]. Tefkat is an implementation of the transformation language proposed by DSTC et al [23] in their response to the OMG's QVT RFP. It is a declarative, logic-based language with a fixpoint semantics. It supports single-direction transformation specifications from one or more source models to one or more target models. The transformation specifications are constructive meaning that they specify the construction of the target model(s). There is currently no support for in-place update of models.

The Tefkat implementation is based on the Eclipse Modeling Framework (EMF) [28] and supports transforming native Ecore models as well as those based on MOF2, UML2, and XMLSchema. It is usable in both standalone form and as an Eclipse plugin with a source-level debugger.

To generate Tefkat rules the TGG plug-in identifies for each object, link and constraint its position (e.g. ≪*left*≫, ≪*map*≫ and ≪*right*≫) and its modifier (e.g. ≪*create*≫ and ≪*delete*≫) and fills based on this information a template for the forward, backward and consistency checking rule. The complete algorithm follows the basic concepts given in Section 3. As an example from the TGG rule 2 the Tefkat rule presented in Fig 8 will be generated.

6 Discussion

With the transformation example given in Section 4 we have shown that triple graph grammars and triple graph grammar rules are suitable to specify simple inter-model-transformations. However, we still see some problems in the application of these triple graph transformation rules to real world transformation problems. First, the success of all transformation languages within the MDA depends mostly on the performance of the application of the transformation rules. It is unacceptable to wait several hours until all rules are applied. The most time consuming task within the application of triple graph grammar rules, is to find all matches of the left hand side (LHS) in an application graph. Consequently, it is necessary to optimise the rule matching algorithms. This can be done e.g. by specifying or identifying an optimal order in which the objects should be matched.

Another important point is an optimal support for an easy specification of transformation rules. This includes a tool that guides the user within the specification without restricting them too much. At this point Fujaba helps a lot. However, it can still be improved. The other aspects that comes to mind when talking about the easy specification of a rule, is a support to reuse and to adapt existing rules. This includes extending and superseding rules as described in [23]. To our current knowledge, there is no theoretical concept for applying inheritance to graph grammar rules.

Finally, current triple graph grammar rules are only suited to model transformations between one source and one target model. To solve this problem in [29], an extension, called MDI-rules, is presented, which allows transformation between N source models and M target models. To provide this possibility, for each additional source or target model an additional graph production rule must be specified. This means, that a 1-to-2 transformation must be specified with quadruple graph grammar rules. However, in most cases these N-to-M transformations can be also specified with a set of 1-to-1 transformation rules, except from the case of model merging (N-to-1), where not only tree-based structures are involved [30].

7 Conclusion and Future Work

This paper describes a possible extension for the current transformation language within the MDA$^{\text{TM}}$ approach. This extension is based on triple graph grammars and triple graph grammar rules, which provide a deep theoretical concept for data integration [9] between different graph-based structures. Thus, they can easily be adapted for model-to-model transformations [26]. An important feature of triple graph grammar rules is the implicit creation of a correspondence graph between the two models. This allows incremental change propagation in case one model evolves. The practical applicability of triple graph grammars for model-to-model transformations is presented with the well-known example of the transformation from an object-oriented class model to a relational database model.

The main benefit of triple graph grammars is the ability to graphically specify transformation rules. However, it needs to be proven that these graphical transformation rules are really easier to specify and to maintain. One possibility to prove this would be with empirical studies. We are currently planning such study with student teams that need to specify and maintain (implement new requirements) a complex model-to-model transformation system.

Besides the benefits of triple graph grammars, we have also discussed (cp. Section 6) the existing problems with applying TGG-rules in real-world model transformation systems. Thereby, especially the extension of the theoretical concepts to allow inheritance and M-to-N transformations with TGG-rules seem to be interesting research topics. With these extensions and optimized algorithms for matching the right hand side of a rule, we think that triple graph grammars can become a useful concept for specifying and applying model-to-model transformations within the model driven engineering paradigm.

References

1. OMG (The Object Managemant Group): MDA specifications, http:// www.omg.org/ mda/ specs.htm. (2002-2004)
2. OMG (The Object Managemant Group): OMG MOF 2.0 query, views, transformations request for proposals (QVT RFP), http://www.omg.org/ techprocess/ meetings/ schedule/ MOF 2.0 Query View Transf.RFP.html or http://www.omg.org/docs/ad/02-04-10.pdf (2002)
3. Gardner, T., Griffin, C., Koehler, J., Hauser, R.: A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard, http://www.omg.org/docs/ad/03-08-02.pdf (2003)
4. Ehrig, H., Pfender, M., Schneider, H.J.: Graph grammars: An algebraic approach. In Book, R.V., ed.: Proceedings of the 14th Annual Symposium on Switching and Automata Theory, University of Iowa, IEEE Computer Society Press (1973) 167–180
5. Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.J., Kuske, K., Plump, D., Schürr, A., Taentzer, T.: Graph transformation for specification and programming. Science of Computer Programming **34** (1999) 1–54
6. Fowler, M.: Refactoring - Improving the Design of Existing Code. Addison Wesley (1999)
7. Van Gorp, P., Van Eetvelde, N., Janssens, D.: Implementing refactorings as graph rewrite rules on a platform independent meta model. In: Proceedings of Fujaba Days 2003. (2003)
8. Mens, T., Demeyer, S., Janssens, D.: Formalising behaviour preserving program transformations. In: Graph Transformation. Volume 2505 of Lecture Notes in Computer Science., Springer-Verlag (2002) 286–301
9. Schürr, A., Winter, A., Zündorf, A.: Graph grammar engineering with PROGRES. In: Proceedings 5th European Software Engineering Conference ESEC. Volume LNCS 989., Springer (1995) 219–234
10. Grunske, L., Geiger, L., Zündorf, A., VanEetvelde, N., VanGorp, P., Varró, D.: Using graph transformation for practical model driven software engineering. In: Model-driven Software Development - Volume II of Research and Practice in Software Engineering, edited by Sami Beydeda and Volker Gruhn, ISBN: 3-540-25613-X. (2005) 91–119
11. Becker, S., Haase, T., Westfechtel, B., Wilhelms, J.: Integration tools supporting cooperative development processes in chemical engineering. In: Proceedings Integrated Design and Process Technology (IDPT-2002), Pasadena, California (2002)
12. Drewes, F., Hoffmann, B., Plump, D.: Hierarchical graph transformation. J. Comput. Syst. Sci. **64** (2002) 249–283
13. Grunske, L.: Automated software architecture evolution with hypergraph transformation. In: 7th International Conference Software Engineering and Application (SEA 03), Marina del Ray, CA, USA (2003) 613–621

14. Grunske, L.: Formalizing architectural refactorings as graph transformation systems. In: Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD05), Towson, Maryland, USA, IEEE Computer Society, IEEE Computer Society (2005) 324–329
15. Baresi, L., Heckel, R.: Tutorial introduction to graph transformation: A software engineering perspective. In: International Conference on Graph Transformation, ICGT, LNCS. Volume 2505 of Lecture Notes in Computer Science., Springer (2002) 402–439
16. Habel, A.: Hyperedge replacement: grammars and languages. Volume 643 of Lecture Notes in Computer Science. Springer-Verlag Inc., New York, NY, USA (1992)
17. Pratt, T.W.: Pair grammars, graph languages and string-to-graph translations. Journal of Computer and System Sciences **5** (1971) 560–595
18. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic approaches to graph transformation I: Basic concepts and double pushout approach. In Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph transformation, Volume 1: Foundations. World Scientific (1997) 163–246
19. Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic approaches to graph transformation II: Single pushout approach and comparison with double pushout approach. In Rozenberg, G., ed.: The Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations. World Scientific (1997) 247–312
20. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae **26** (1996) 287–313
21. FUJABA: (Fujaba homepage http://www.fujaba.de/)
22. Schürr, A.: Specification of graph translators with triple graph grammars. In: Proceedings 20th Workshop on Graph-Theoretic Concepts in Computer Science. (1994) 151–163
23. DSTC/IBM/CBOP: Second revised submission for MOF 2.0 Query / Views / Transformations RFP, http://www.omg.org/ docs/ad/04-01-06.pdf (2004)
24. QVT-Partners: Revised submission for MOF 2.0 Query / Views / Transformations RFP, http://www.omg.org/docs/ ad/03-08-08.pdf (2003)
25. Jahnke, J.H.: Management of Uncertainty and Inconsistency in Database Reengineering Processes, Ph.D Thesis Uni Paderborn (2002)
26. Kindler, E., Rubin, V., Wagner, R.: An Adaptable TGG Interpreter for In-Memory Model Transformation. In Schürr, A., Zündorf, A., eds.: Proc. of the 2nd International Fujaba Days 2004, Darmstadt, Germany, University of Paderborn (2004) 35–38
27. DSTC: Tefkat: The EMF Transformation Engine, online documentation. (`http://www.dstc.edu.au/tefkat/`)
28. Merks, E., Eliersick, R., Grose, T., Budinsky, F., Steinberg, D.: The Eclipse Modeling Framework. Addison Wesley (2003)
29. Königs, A., Schürr, A.: Multi-domain integration with mof and extended triple graph grammars. In: in Proceedings of the Dagstuhl Seminar 04101, Language Engineering for Model-Driven Software Development J. Bzivin (Univ. Nantes, FR), R. Heckel (Univ. Paderborn, DE), Dagstuhl (2004)
30. Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. Software Eng. **28** (2002) 449–462

Horizontal Transformation of PSMs

Jamal Abd-Ali and Karim El Guemhioui

Department of Computer Science and Engineering,
University of Quebec in Outaouais (UQO),
C.P. 1250 succ. Hull, Gatineau (Quebec), H3A 2N4 Canada
{abdj01, karim}@uqo.ca

Abstract. In this last decade component technology has known a fast expansion, initially with EJB (Enterprise JavaBeans), and more recently with .NET. Many companies would probably not hesitate to embrace the new technological wave of .NET components if they could recover (part of) their investment in EJB. As it is more than probable that existing EJB applications do not have models independent of the technological platform (PIM), we propose a horizontal migration path between these two technologies by the definition of a transformation which converts a model specific to EJB (PSM) into a model specific to the .Net components. Since metamodels of these two technologies are essential to the definition of our transformation, we use the EJB metamodel adopted by the OMG and, for the .NET components, we propose a metamodel of our vintage. The transformation is written in a well-defined language derived from a submission in response to the RFP issued by the OMG to standardize the transformation language QVT. The feasibility of the idea is illustrated with an example.

Keywords: Model Transformation, Metamodeling, MDA, MDE, Horizontal Transformation, EJB to.NET.

1 Introduction

The development of modern applications is not only characterized by the reliance on component technology, but also by a substantial modeling activity that goes beyond the mere documentation of the system under development, and becomes central to production. This is achieved by the resort to code generators, and more recently, to comprehensive conceptual frameworks based on the Model Driven Architecture (MDA) [8, 5] defined by the Object Management Group (OMG) [11]. MDA and, more generally, Model Driven Engineering (MDE) approaches advocate neutral design from any technology of the moment, before the derivation of specific models for targeted technological platforms and the automatic generation of code from these latter models.

Now, each time that a new technology emerges by offering solutions to existing problems and a set of supporting tools, we observe a migration towards the new technology. The recent .Net components technology does not seem to be an exception to this rule. While the EJB technology has a recognized metamodel [9], the .Net components technology, to the best of our knowledge, does not have a published one which we could use to define models specific to this platform (PSMs). Furthermore, such a metamodel would allow the definition of transformation rules for an automatic conversion between PSMs, avoiding the costly adaptation of existing applications to the newest platform.

In this paper, we propose a horizontal migration path between these two widespread component technologies, by the definition of a transformation that makes it possible to shift from a model specific to EJB (PSM) to a model specific to the .Net components. In absence of a recognized metamodel for the .NET components, we use our own metamodel [1], certainly perfectible and that we are ready to amend or replace if a better proposal becomes available.

2 The EJB Metamodel

The work of several pioneer companies in object oriented distributed computing gave rise to a metamodel of the Enterprise JavaBeans architecture [9] which was standardized by the OMG. The core of this metamodel is represented in figure 1. Every EJB component is derived from EnterpriseBean which is part itself of an EJBjar element. The latter is the root of a deployment descriptor containing, among, other things, the required information for the deployment and management of the EJB component. The Assembly element embodies the deployment descriptor by organizing the information it contains according to a structure defined in the specification of this technology.

A Session type component offers services to clients, via its public interface, and defines methods implementing domain specific functionalities. An entity type component models a domain concept and offers a service for managing the persistence of the states of its instances.

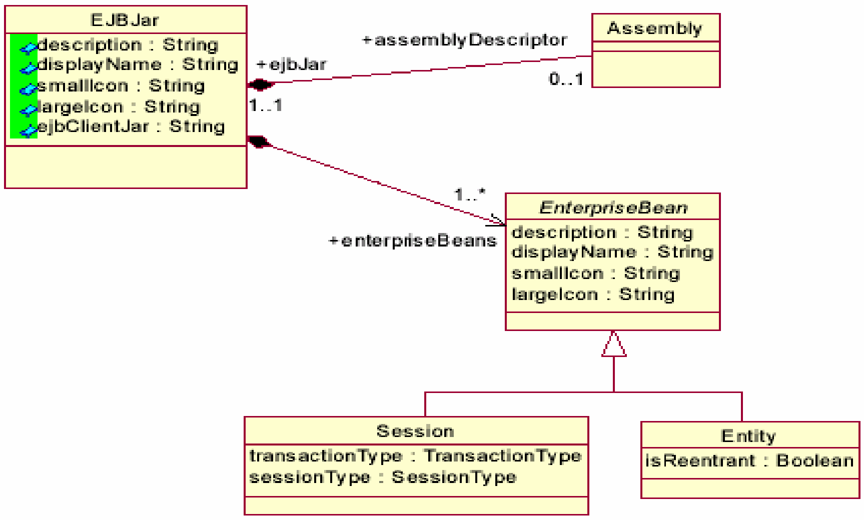


Fig. 1. Main diagram of the EJB metamodel

The EJB metamodel encompasses other diagrams and relies on the Java metamodel. We refer the interested reader to the official standard [9] and the latest specification [3] for further details.

3 A .Net Components Metamodel

The proposed metamodel has been devised with the main concern of capturing the essential concepts and semantics of the core .Net components technology. A detailed presentation and discussion of the metamodel can be found in [1]. In this paper, we limit ourselves to a quick overview of the diagrams of our metamodel relevant to the understanding of the transformation discussed later.

3.1 Main .NET Components Diagrams

Figure 2 shows that every class implementing a .NET component must inherit from MarshalByRefObject (MBR), and that it is generally associated to a host application, a client application, and possibly a COM+ application offering a set of services.

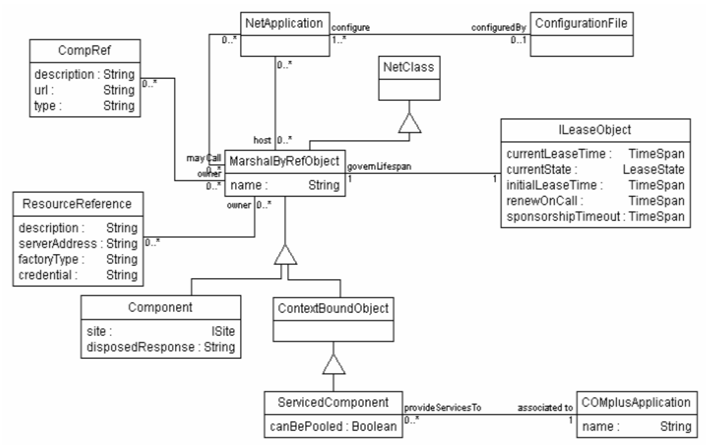


Fig. 2. The .NET Components metamodel main diagram

Not shown in this figure but useful for understanding the transformation of section 4, the metamodel comprises also an element named Attribute that represents the Attribute concept in the .NET world. NetClass is related to Attribute with the role name “mark”. The NetDefinedAttribute and all the metamodel elements whose names end with “Attribute” inherit the Attribute element.

Figure 3 summarizes the main COM+ services offered to any .NET component derived from the ServicedComponent class. A detailed explanation of the COM+ services and the .NET component technology can be found in [6], [7], and [16].

The proposed metamodel remains simple and yet comprehends the key elements of the .Net component technology. It allows to describe the implementation of .NET component based applications, as well as transformation rules targeting this technology. It could also be used for a quick introduction to this technology, from a software architecture perspective.

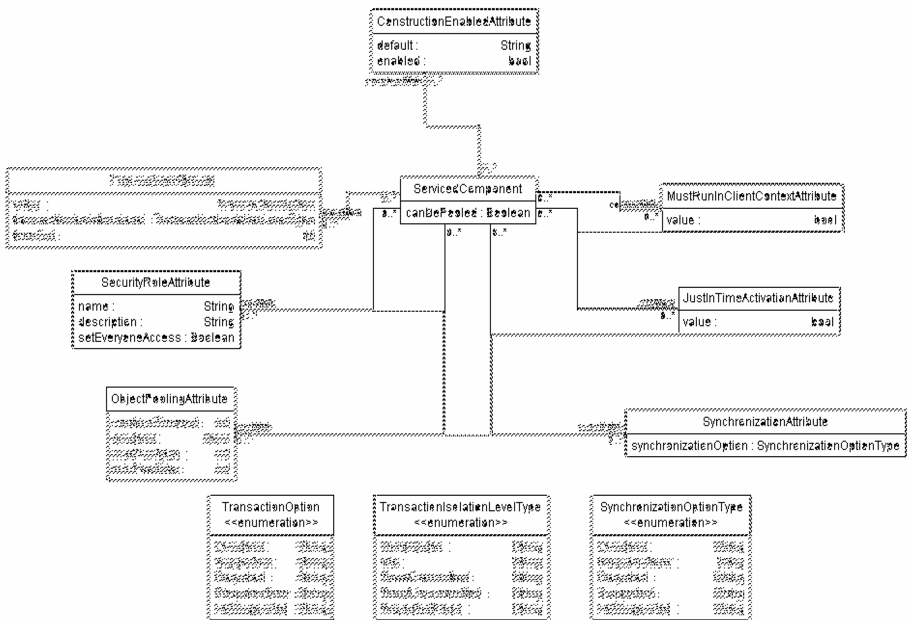


Fig. 3. COM+ Services diagram of the .NET Components metamodel

3.2 Preliminary Considerations on Transformation Characteristics

A model transformation is a process that enables the conversion of a model into another one [8]. From an MDA viewpoint, a transformation is meant to be automated; it consists of a set of rules, and must be written in a well-defined language that a tool can compile and execute. Therefore, this automation implies that the models must be written in a language that ultimately can be processed by machines.

A transformation language is characterized by the way its rules are applied, which entails that the following elements be specified [2]:

- The order in which the different rules are applied;
- Rule composition and structuring;
- The relationship between source and target models (same, different);
- The ability to restrict the transformation to a part of the source model;
- The traceability that records links between source and target elements;
- Directionality specifying whether the transformation can be executed in both directions between source and target models;
- Parameterization allowing tuning and configuration of transformation rules.

We also need to closely examine the pivotal element of any transformation, which is the rule. It is generally an expression of mappings between a fragment of the source model and a fragment of the target model; and it is characterized by [2]:

- The rule application strategy to determine which match to process first, when we have more than one match for a rule;
- Variables usage, their type and visibility;
- How to specify the matching model fragment for a rule using direct manipulation, graph theory, etc.;
- Execution logic: imperative or declarative;
- Directionality, traceability, and parameterization at the rule level.

4 The Transformation

For both discussed technologies (EJB and .NET), each element of a PSM is an instance of a unique element of its metamodel. Hence, we will specify each element involved in a transformation rule, by referring to its type in the corresponding metamodel and by imposing a filter condition on the properties of this element and on the PSM model to which it belongs.

4.1 Expression Language

The transformation that converts an EJB PSM into a .NET PSM is made of a set of rules. To write these rules, we use the model transformation language OpenQVT [10] as implemented by the actual syntax of the ATL tool [15] which, in its turn, supports the standardized Object Constraint Language [12]. The following ATL syntactic details will help in the reading of the proposed rules:

- The ATL compiler is case sensitive regarding source code and file names
- The name of a rule must be unique within its scope
- The symbol ! denotes the scope operator ; the symbol -- denotes a comment
- A value assignment is done via the left arrow operator <-
- Two assignment values in a row are separated by a comma,
- The symbol # at the beginning of a string indicates that the string represents an enumeration member
- To navigate from an element to its attribute, write the element name followed by a dot, then the attribute name
- To express string concatenation use the operator +
- Literals are expressed between quotes “ ”
- Filter conditions are expressed in OCL

To express the definition of a transformation rule, we use the following syntax:

```
rule R {
      from      s : name-meta-source!el-s (cond)
      to        t : name-meta-target!el-t
       (-- sequence of value assignments to populate
                    -- newly created element
                    -- e.g.; title<-s.title,name<-s.name + “new” )
        }
```

With:

R: name of the rule.

name-meta-source : name of metamodel of source model.

el-s: name of a metamodel element. Instances of this element will be, in turn, subjected to the rule R.

s: local name of el-s instance (being processed by R) occurring in the source model.

cond : filtering condition on source instances input to the rule R.

name-meta-target :name of metamodel of target model.

t: local name of instance created as output to the rule R.

el-t: name of metamodel element to instantiate in target model, whenever an instance s occurs in the source model.

Besides rules, one can specify actions to be triggered at the beginning of the transformation or of some rule. Such actions are defined in a block of code with an *init* header. Note that *init* as well as other elements of OpenQVT are currently not directly supported in ATL.

The following notations will help simplify the expression of our transformation rules:

EJB : The EJB metamodel.
Java : The Java metamodel.
Net : The .Net Components metamodel.
in : the source model.
out : the target model.

In OpenQVT, the "Net::UseDefaultValue := true" declaration means that the creation of an element causes the creation of all of its owned elements initialized to their default values. This declaration will be implicit in the formulation of our transformation rules.

If A and B are two metamodel elements related by an association, the navigation at the model level, from an instance a of A to an instance b of B, uses the name of the role played by B in the association. If no role name is explicitly mentioned in the association, the name B is then used. Hence, a.B = a.roleOfB = b.

Moreover, the names of the instances following the keywords *from* and *to* will be used as names of variables visible inside the same rule they are declared in.

These last two syntactic uses are not supported by ATL, but facilitate the reference to model elements within the same rule, and the efficient formulation of the transformation rules.

4.2 Mapping of Java Primitive Elements

We recall that the EJB metamodel builds upon the Java metamodel, and therefore one must define .NET mappings for primitive Java elements (Class, Field, Method, Interface, Parameter), without getting lost in subtleties inherent to the respective technologies and beyond the scope of this work. For simplicity and due to the semantic coincidence of the concepts of class, field, type, and parameter in the two worlds (Java and .NET), we will be content with the Java metamodel to provide the expression language for their counterparts in .NET. Though, to avoid any confusion, the Java class type will be called JavaClass, and the .NET class type will be called NETClass.

Furthermore, the Java primitive types will keep the same names in .NET, except for the Java boolean type which is called bool in .NET. We also adopt the Java modifier default values for the Field, Method, and Class elements of .NET.

4.3 Transformation Rules

The rules are briefly explained in natural language, and then formulated in the ATL syntax previously introduced. In our explanations in natural language, we use the symbol = to mean assignment of value as well as a reference to an element of the source model implying an equality with its mapping by the transformation rule. We have defined a total of 13 transformation rules, but for lack of space, we only show the rules involved in the example at the end of the paper.

Rule 1. COMplusApplication - If the source model contains at least one instance of type EJBJar, we create in the target model one instance of COMplusApplication, we call Comp, with:

- The attribute name is set to “Comp”
- We create an instance appAct of the ApplicationActivationAttribute element and set its activationOption attribute to Server.
- We create an instance *as* of Assembly ; *as* is associated to Comp and appAct.
- We create an instance appAc of ApplicationAccessControl, associate it to *as*, and set the following attributes:
  - Enabled = true
  - AccessCheckLevel = ApplicationComponent
  - Authentication = Connect
  - ImpersonationLevelOption = Default

```
rule R1 {
    from jar : EJB ! EJBJar  ( jar = EJBJar.allInstances( ) ->asSequence( )->first( ))
    -- the filter is set so as to select only the first instance of  EJBJar
    to as : Net  ! Assembly
    to  Comp : Net ! COMplusApplication  (name <- “Comp”, Comp.Assembly <-as) ,
    to appAC : Net ! ApplicationAccessControl  (enabled <- true,
       accessCheckLevel <- #ApplicationComponent ,
     authentication <- #Connect, impersonationLevelOption <- #Default , Assembly <- as),
    to   appAct :  Net!ApplicationActivationAttribute  (activationOption  <-  #Server,  Assembly
    <- as)
    }
```

Rule 2. Application, LifeTime, Channels, Channel and Service – If the source model contains at least one instance of type EJBJar, we create and initialize in the target model, instances of NetApplication, ConfigurationFile, Application, LifeTime, Channels, Channel and Service.

```
rule R2 {
    -- create instances of Application, NetApplication, and ConfigurationFile
    from jar : EJB!EJBJar  ( jar = EJBJar.allInstances( )->asSequence( )->first( ))
    -- the filter used allows to select only the first instance of  EJBJar, if it exists.
    to app : Net!NetApplication  ( ) ,
    to conf : Net!ConfigurationFile  (configure <- app),
     to appElement : Net!Application  (owner <- conf),
    to chs : Net!Channels   (owner <- appElement),
    to ch : Net!Channel  (owner <- chs, ch.ref <- "tcp", ch.port <- 8010),
    -- 8010: arbitrary choice; can be changed by whoever executes
    -- the transformation
    to lf : Net!LifeTime  (lf. LeaseTime <- "1 H",      -- 1 H means one hour
    -- arbitrary choice
            lf.SponsorshipTimeout <- "1 H",
            lf.RenewOnCallTime <- "1 H",
            lf.LeaseManagePollTime <- "1 H",
            owner <- appElement),
    to serv : Net!Service  (owner <- appElement)
    }
```

Rule 3. Bean Session stateful or stateless – For each instance S of EJB Session, we create:

- An instance SC of .NET ServicedComponent, with:
  - name of SC corresponding to S.remoteInterface.name
  - If S**.**sessionType = Stateful, the attribute CanBePooled of SC is set to false
  - If S.sessionType = Stateless, the attribute CanBePooled of SC is set to true
  - SC.associatedto = the unique instance Comp of COMplusApplication already created
  - SC.netApplication = the unique instance of NetApplication
- An instance opa of ObjectPoolingAttribute, with:
  - enabled = true
  - creationTimeout = 5
  - maxPoolSize = 10
  - minPoolSize = 0
  - opa.mark = S
- An instance act of Activated and we associate it to the unique instance of Service already created, with:
  - act.Service = the unique instance of Service already created by rule 2
  - act.ref = "tcp://localhost:8010" - This arbitrary value can be adjusted by whoever executes the transformation.
  - act.type = SC.name + "," + SC.name

- o An instance syn of SynchronizationAttribute, with :
    - syn.synchronizationOption = #Required
    - syn.mark = SC – mark being the role name of any element with regard to the attribute that affects it.

Below, the formulation of this rule and of the *pooling* function it uses:

```
        -- We start by defining an operation on the instances of Session.
        helper context EJB!Session
        def : pooling ( ) : Boolean = if sessionType = #Statless true else false;

rule R3 {
    from S : EJB!Session
    to SC : Net!ServicedComponent ( CanBePooled <- S.pooling( ), name <-
    S.remoteInterface.name,
        SC.associatedto = l'unique instance Comp, SC.netApplication <-
        NetApplication.allInstances()),
    to opa : Net!ObjectPoolingAttribute ( opa.enabled <- true, opa.creationTimeout <- 5,
        opa.maxPoolSize <- 10, opa.minPoolSize <- 0, opa.mark <- S)
    to act : Net!Activated (owner <- Service.allInstances ( ),
    -- Only one instance of Service is created
        ref = "tcp://localhost:8010", type <- S.ejbClass.name + "," + S.ejbClass.name ),
    to syn : Net!SynchronizationAttribute ( syn.synchronizationOption <- #Required )
    }
```

Note that we created a .NET instance of ObjectPoolingAttribute for an EJB instance S regardless of its sessionType. If the latter is stateful, this creation is not necessary. To avoid this systematic creation, we can resort to *helpers* or we can define two different rules selectively targeting the Session instances according to the value of their sessionType attribute.

Rule 4. ContainerManagedEntity – In order to create in .Net a pattern that represents an Entity bean whose persistence is managed by the EJB *container*, we apply a rule that creates for each instance of ContainerManagedEntity, a component SC of type ServicedComponent which is at the disposal of calling clients, and which declares a serializable class (class name with suffix "_Serializable") holding all the fields whose persistence we want to manage. An instance *att* of .NET Attribute will contain the information on the primary key fields, and will be associated to the SC component to supply it with all this information.

After that, the management of persistence becomes easy by using a DBMS (e.g.; SQLServer) supporting the persistence management of serializable class objects in XML format.

The mapping detailing this rule is as follows:

For each instance *ent* of EJB ContainerManagedEntity, we map:

- o An instance ESerial of Netclass containing the persistent fields, with:
    - NetDefinedAttribute.name = "Serializable"
    - name = ContainerManagedEntity.remoteInterface.name + "_Serializable"

- An instance SC of ServicedComponent associated to COMplusApplication and NetApplication instances, with:
  - name = ContainerManagedEntity.remoteInterface.name
  - CanBePooled = true
  - Maping methods of remote and home interfaces
  - We create an instance act of .NET Activated with :
    - act.Service = the unique instance of Service already created by rule 2
    - act.ref = tcp://localhost:8010
    - act.type = SC.name + ",'' + SC.name
- An instance opa of ObjectPoolingAttribute with :
  - enabled: true
  - creationTimeout = 5
  - maxPoolSize =10
  - minPoolSize =0
  - opa.mark = SC
- An instance kNames of .NET Attribute with:
  - Creation of an instance f of .NET Field
  - f.name = "KeyFieldsNames"
  - f.type = String
  - The created field f will be used to store the set of all the names of the primary key fields of the entity. This set of names can be represented by the value:

  ent.keyFor ->iterate(att : Field, acc :String ="" |acc + att.name + ",")
  - kNames.mark =SC

    We have created an Attribute which marks SC and supplies it with the names of the primary key fields separated with commas. An instance syn of .NET SynchronizationAttribute with:
  - syn.synchronizationOption = Disabled if syn.isreentrant = true ; otherwise is will be equal to #Required.
  - syn. mark = SC

Rule 5. References to other components – For every instance eref of EJBRef, we create an instance cref of .NET CompRef, with:

- the corresponding descriptions
- cref.type = eref.remote

We associate eref to the SC instance and specify that:

- CompRef.owner maps to cref.ejb

Other rules: The complete transformation definition comprehends many more rules that handle other elements and concepts of the EJB metamodel. However, in the context of this work, we voluntarily limit our explanations to the above shown main set of rules that illustrate our approach.

5 Discussion

When writing our transformation rules, we were aware of the fact that an EJB bean can never become a straight .NET component. We rather tried to produce a target .NET PSM that permits to meet the functional and nonfunctional requirements of the system described in the source PSM. Divergences between source and target must be confined to details collapsed in the models. We set forth the two following criteria to guide us in the development of the transformation rules.

- Do not to alter the business logic conveyed in the source PSM. Indeed, we can easily notice that the EJB metamodel deals primarily with the structural part of the business model, leaving aside implementation details. Hence, our transformation produces a PSM which preserves for each component the fields, interfaces, and references to other components or resources.
- Ensure that the target model represents .Net components that benefit from middleware services comparable with those provided by the EJB model.

Table 1. Main EJB technological concepts and corresponding elements in .Net

EJB Concepts		**Corresponding Elements in .Net Components Metamodel**
Remote access	A component passes a reference to a distant caller which then disposes of a proxy that appears like a local instance of the component.	MarshalByRefObject
	A component has a JNDI name.	Service, Channel, Activated, WellKnown
Security	Role based security with checking at the application, component, method, and interface level.	SecurityRoleAttribute can be associated to different elements.
Instance management	Pooling, passivation of inactive instances, lifetime.	ObjectPoolingAttribute, JITA, ILeaseObject, LifeTime, SingleCall, Singleton
Transactionnel behavior management	There are several choices to determine how to reconcile between the transactional context of the client and that of the component.	TransactionAttribute and its attributes.

Synchronization and reentrance	Container guarantees a queuing of the calls to a session instance. Reentrance is not recommended but always allowed.	SynchronizationAttribute and the allowed options via its synchronizationOption attribute. Reentrance can be allowed by setting the value of synchronizationOption to disabled. The resulting lack of synchronization is worse than reentrance with respect to data consistency that can be controlled at the DBMS level.
Persistence	ContainerManagedEntity allows the storage of its attributes on a permanent support by using a service offered by the container and transparent to the developer	The DefinedNetAttribute whose name is Serializable represents an attribute supporting the serialization of objects, in XML format. On the other hand, SQLServer allows requests or updates by handling serialized objects in XML format. Thus, we always end up with persistence controlled by program, but which directly handles the storage of the instances of the serializable class associated to the ServicedComponent.

Note also that we limited ourselves to a target model comprising only one .Net application and one COM+ application, in order to concentrate on the component characteristics, regardless of the site in which it is running or the number of applications that can reuse it.

The target model does not store EJB transaction management information if the instances of MethodTransaction (related to an EntrepriseBean via MethodElement) do not share the same transactionAttribute value which means that the methods of the same bean have different managements of their transactional behavior. This is due to the fact that the management of transactions in the .NET metamodel is configurable at the component level rather than at the method level.

A developer must also resort to tricks to overcome the fact that a class derived from ServicedComponent does not support a constructor with parameters. For example, we can take advantage of the *create* method in the *home* interface of an EJB bean entity and map it, via the transformation, to a .NET initialization method accepting parameters.

The point is that despite the restrictions in our current definition of the transformation, it is always possible to circumvent them or deal with them via a tool which supports a parameterization of the transformation rules and which offers a user-friendly interaction with the user.

Finally, we need to raise the issue of the reverse transformation from a .NET PSM to an EJB PSM. Obviously, our rules are far from being symmetrical to support a straightforward deduction of their inverse, especially when they associate one EBJ element to several .NET elements. Furthermore, in our initial attempts to define a reverse transformation, we experienced a significant loss of information due mainly to the fact that the standardized EJB metamodel does not preserve the fine granularity of the abstracted EJB services (e.g.; timeOut, poolSize in EnterpriseBean container).

6 Testing the Transformation

We illustrate the workings of our transformation through a model representing an ecommerce application similar to the broadly published Pet Store example. Our example, borrowed from [4], consists of a virtual web shop selling goods that clients can order.

The EcommerceSession bean is the EJB component ready to handle client application users. It authenticates a user and offers several services allowing the latter to fill her shopping cart and order the goods in it. This bean being stateful, its object instance keeps the information about its interactions with users until the end of its lifetime. The bean Person assists our Session bean in managing user data persistency. It allows the creation of new instances and of requests to existing ones. To manage the persistency of information related to orders, we can use Entity beans representing the orders, the order lines, the goods data, etc. However, to stay focused on the EBJ metamodel transformation rather than on the accuracy of the application, we limited ourselves to a Session bean interacting with a client and using another Entity bean (implicitly assuming a direct interaction between the Session bean and a DBMS).

Figure 4 shows the EJB PSM source model. With respect to notation, an instance of a Method or Field element is represented in the same way we represent an operation in UML. An instance of an element called X in the metamodel is represented as a UML class with the stereotype <<X>>.

Figure 5 shows the .NET PSM target model produced by our transformation. The circled numbers on the figure indicate the name of the rule applied to create the annotated element. Notice that the output model can be shown in one diagram but for paper size and readability we choose to divide it in three diagrams.

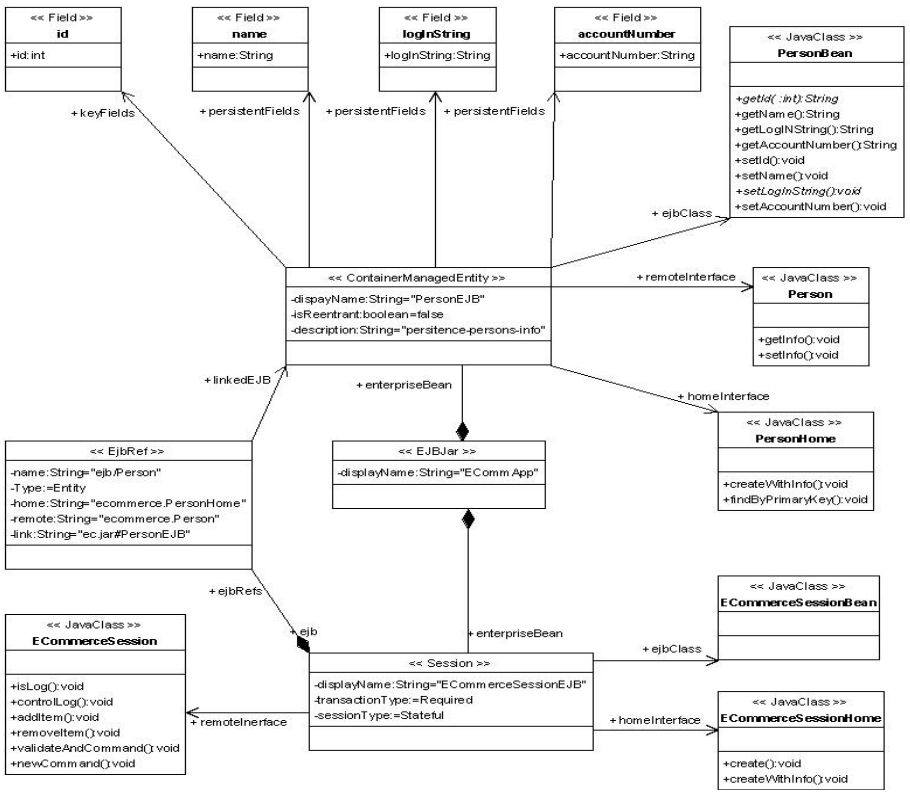
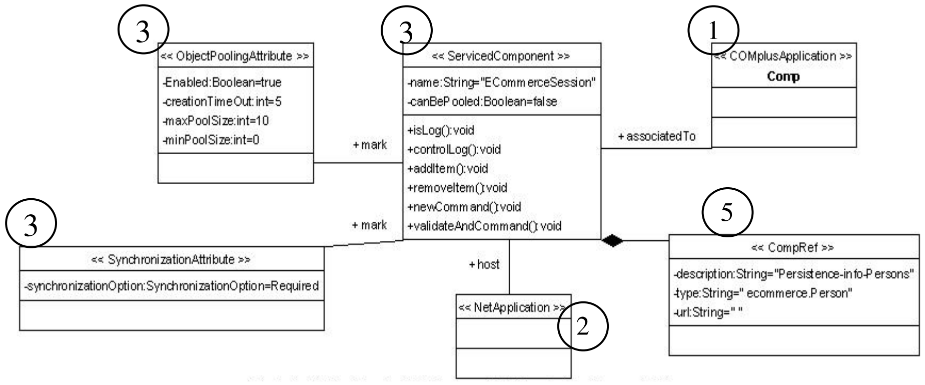


Fig. 4. EJB PSM for the ecommerce application



3
<< ObjectPoolingAttribute >>
-Enabled:Boolean=true
-creationTimeOut:int=5
-maxPoolSize:int=10
-minPoolSize:int=0
3
<< ServicedComponent >>
-name:String="ECommerceSession"
-canBePooled:Boolean=false
+isLog():void
+controlLog():void
+addItem():void
+removeItem():void
+newCommand():void
+validateAndCommand():void
1
<< COMplusApplication >> Comp
+associatedTo
+mark
+mark
3
<< SynchronizationAttribute >>
-synchronizationOption:SynchronizationOption=Required
+host
<< NetApplication >>
2
5
<< CompRef >>
-description:String="Persistence-info-Persons"
-type:String=" ecommerce.Person"
-url:String=" "

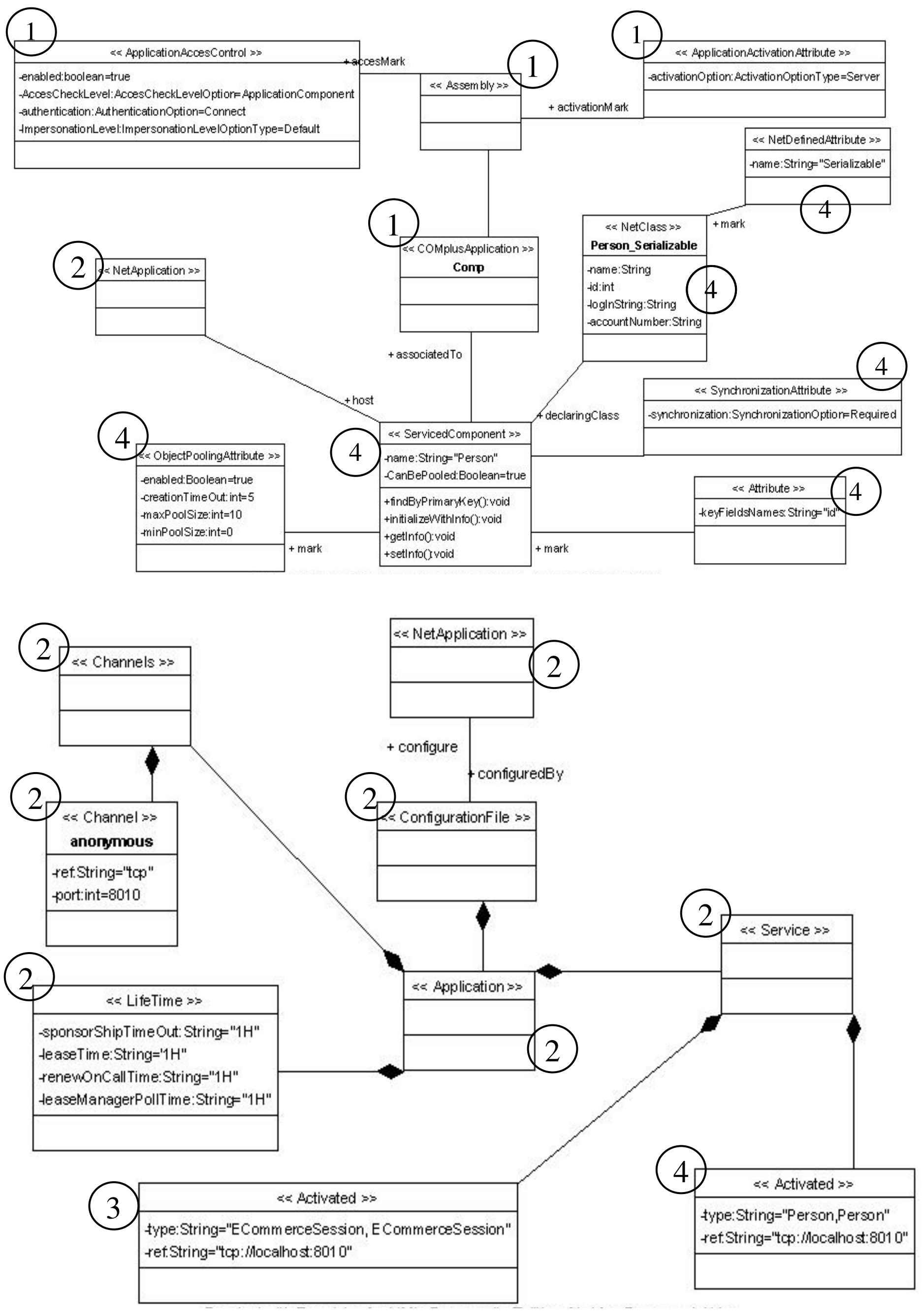

1
<< ApplicationAccesControl >>
-enabled:boolean=true
-AccesCheckLevel:AccesCheckLevelOption=ApplicationComponent
-authentication:AuthenticationOption=Connect
-ImpersonationLevel:ImpersonationLevelOptionType=Default
+accesMark
1
<< Assembly >>
+ activationMark
1
<< ApplicationActivationAttribute >>
-activationOption:ActivationOptionType=Server
<< NetDefinedAttribute >>
-name:String="Serializable"
4
+mark
<< NetClass >>
Person_Serializable
-name:String
-id:int
-logInString:String
-accountNumber:String
4
1
<< COMplusApplication >>
Comp
2
<< NetApplication >>
+associatedTo
+host
+declaringClass
4
<< SynchronizationAttribute >>
-synchronization:SynchronizationOption=Required
4
<< ObjectPoolingAttribute >>
-enabled:Boolean=true
-creationTimeOut:int=5
-maxPoolSize:int=10
-minPoolSize:int=0
+ mark
4
<< ServicedComponent >>
-name:String="Person"
-CanBePooled:Boolean=true
+findByPrimaryKey():void
+initializeWithInfo():void
+getInfo():void
+setInfo():void
+ mark
<< Attribute >>
-keyFieldsNames:String="id"
4
2
<< Channels >>
<< NetApplication >>
2
+ configure
+ configuredBy
2
<< Channel >>
anonymous
-ref:String="tcp"
-port:int=8010
2
<< ConfigurationFile >>
2
<< Service >>
2
<< LifeTime >>
-sponsorShipTimeOut:String="1H"
-leaseTime:String="1H"
-renewOnCallTime:String="1H"
-leaseManagerPollTime:String="1H"
<< Application >>
2
3
<< Activated >>
-type:String="ECommerceSession, ECommerceSession"
-ref:String="tcp://localhost:8010"
4
<< Activated >>
-type:String="Person,Person"
-ref:String="tcp://localhost:8010"

Fig. 5. .Net PSM produced by the transformation (three diagrams)

7 Conclusion

The proposed .Net Components metamodel can be considered a first step towards a standardized metamodel comparable with that of Enterprise JavaBeans. This standardization of metamodels is crucial to fully benefit from a model driven engineering approach, since the latter provides the appropriate language for the writing of PSMs essential to the definition of model transformations.

We believe that the definition of a horizontal transformation of PSMs between two widespread technologies (namely, EJB and .NET) is beneficial to developers wishing a diversification of their technology or migration to the other technology at a cut-price cost. Hopefully, this work should open the door to other initiatives of metamodeling and transformation definitions targeting other technologies.

A logical continuation of this work is the development of a code generator based on the .NET metamodel. Another extension will be the definition of the two vertical transformations which will make it possible to convert a PIM written in an EDOC UML profile [13] into PSMs specific to EJB and .Net, respectively.

Another interesting issue to investigate would be to compare and contrast our horizontal transformation from PSM to PSM with an approach in which the original PSM is first reversely transformed into a PIM, then this latter transformed into the target PSM.

Finally, the contribution of metamodeling to the software development process is still far from its full potential, partly due to a lack of tools and standards. We hope that future work will provide relevant technologies metamodels as well as integrated development environments open to these metamodels, and supporting the edition and automated transformation of models.

References

1. Abd-Ali, J., K. El Guemhioui. "An MDA-Oriented .Net Metamodel", 9th IEEE International EDOC Conference (EDOC 2005), Enschede, The Netherland, Sept. 2005.
2. Czarnecki, K., S. Helsen. "Classification of Model Transformation Approaches". 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, Anaheim, CA, October 2003.
3. EJB Specifications from Sun http://java.sun.com/products/ejb/docs.html
4. Fauré, A., N. Soukouti. EJB 2.0 Mise en oeuvre. Dunod. 2002.403 p.
5. Kleppe, Anneke, Jos Warmer, Wim Bast. MDA Explained - The Model Driven Architecture : Practice And Promise. Addison-Wesley, 2003. 170 p.
6. Lowy, Juval. COM and .NET Component Services. O'Reilly, 2001. 384 p.
7. Lowy, Juval. Programming .NET Components. O'Reilly, 2003. 459 p.
8. OMG: MDA GUIDE Version 1.0.1 document number omg/2003-06-01 available at http://www.omg.org/docs/omg/03-06-01.pdf
9. OMG. Metamodel and UML Profile for Java and EJB Specification. February 2004. Version 1.0, formal/04-02-02. An Adopted Specification of the Object Management Group, Inc.
10. OMG: MOF 2.0, Query/Views/Transformation. ad/2002-04-10, Revised Submission, Version 1.0, 2003/08/18, OpenQVT; available at http://www.omg.org/docs/ad/03-08-05.pdf

11. OMG : Object Management Group. www.omg.org
12. OMG : OCL Response to the UML 2.0 OCL RfP (ad/2000-09-03). Revised Submission, Version 1.6. January 6, 2003. OMG Document ad/2003-01-07; available at http://www.omg.org/docs/ad/03-01-07.pdf
13. OMG: UML Profile for EDOC. Available at http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML_for_EDOC
14. Sun Microsystems, Enterprise JavaBeans. http://java.sun.com/products/ejb/
15. The ATL language definition page web - Available at http://www.sciences.univ-nantes.fr/lina/atl/atlProject/languageDefinition/
16. The Microsoft Developer Network (MSDN). Available at http://msdn.microsoft.com/library/default.asp

Automatic Support for Traceability in a Generic Model Management Framework*

Artur Boronat, José Á. Carsí, and Isidro Ramos

Department of Information Systems and Computation,
Polytechnic University of Valencia, Camí de Vera s/n, 46022 Valencia-Spain
{aboronat, pcarsi, iramos}@dsic.upv.es

Abstract. In a MDA process, software artifacts are refined from the problem space (requirements) to the solution space (application). A model refinement involves the application of operators that perform tasks over models such as integrations and transformations, among others. We are working on a model management framework, called MOMENT (MOdel manageMENT), where model operators are defined independently of any metamodel in order to increase their reusability. This approach also increases the level of abstraction of solutions of this kind by working on models as first-class citizens, instead of working on the internal representation of a model at a programming level. In this context, traceability constitutes the mechanism to follow the transformations carried out over a model through several refinement steps. In this paper, we focus on the generic traceability support that the MOMENT framework provides. These capabilities allow the definition of generic complex operators that permit solving specific problems such as change propagation.

Keywords: Model-Driven Architecture, Model Management, traceability, software maintenance.

1 Introduction

Traceability is an important issue in environments where there is a process chain. In these cases, information about each step in the chain may be stored for further processing. For example, in the automotive industry, traceability makes recalls possible; in the food industry, it contributes to food safety; in the Software Engineering field, it provides support for requirements validation and improves the quality of the software development process.

In any scenario of the Software Engineering field, there is a manipulation of a software artifact. The capability of describing and querying the manipulation that has been performed on a specific artifact might be relevant to correlated tasks. However, traceability still remains in the background when software engineering problems are solved. It is often misunderstood and burdensome due to the lack of tools that provide full automatic support for it [1, 2].

* This work was supported by the Spanish Government under the National Program for Research, Development and Innovation, DYNAMICA Project TIC 2003-07804-C05-01.

In the Model-Driven Architecture initiative [3] (MDA), a software artifact is viewed as a model. Typical tasks, such as code production, integration of applications, interoperability between applications, are performed on models directly. This allows the user to work at a conceptual level, and makes the identification of the elements needed to automate these tasks easier. These tasks are pervasive in many scenarios and are usually solved in an ad-hoc manner.

Following this model-driven approach, a new discipline, called Model Management, was proposed in [4]. This discipline considers models as first-class citizens and provides a set of generic operators to deal with them: *Merge*, *Diff*, *ModelGen*, etc. These operators provide a reusable solution to the tasks described above so that the user deals directly with models, rather than working on the internal representation of a model at a programming level. Several approaches to this discipline [5, 6] specify operators that are based on mappings to deal with models. A mapping is a relationship between an element of a domain model and an element of a range model that indicates that they represent the same element in different models. This means that mappings between two models must be explicitly defined in order to apply an operator to them.

Using our experience in applying the algebraic specification formalism to solve actual software engineering problems [7, 8], we are working on a model management framework called MOMENT (MOdel manageMENT). In our approach, we represent the relationships between two models in an implicit manner by means of an equivalence morphism that is defined between two metamodels. Our approach describes equivalence relationships between two models from a more abstract and reusable point of view. However, explicit mappings between two models are also beneficial when there is no definition of the equivalence morphism between two metamodels. We refer to mappings of this kind as traceability links.

In this paper, we focus on the automatic traceability support that is provided by our framework from a generic point of view. We define what a traceability model is in this setting and how they can be used to provide traceability support in many different scenarios: requirements, workflows, ontologies, etc. We also show how traceability support contributes to automate solutions such as the software maintenance.

The structure of the paper is as follows: Section 2 reviews the traceability management studies that have been performed in the Requirements Engineering field; Section 3 provides an example to illustrate the use of traceability; Section 4 provides an overview of our model management approach; Section 5 details the generic traceability support that is provided in our framework; Section 6 solves the problem of the case study with our model management operators; Section 7 presents some related work; finally, Section 8 summarizes the main features of our approach.

2 The Traceability Problem in Requirements Engineering

In Requirements Engineering, the IEEE Guide to Software Requirement Specifications [9] indicates that a software requirements specification is traceable if the origin of each requirement is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation.

Based on this definition, Gotel and Finkelstein [10] described requirements traceability as the ability to describe and follow the life of a requirement, both forwards and backwards, through all the refinement steps in a software development process. Queries of this kind allow the user to know what refinements have been applied to a requirement (in a forward direction), and permit the identification of a requirement from a more specific software artifact such as code (in a backward direction). Therefore, traceability can be used for requirements validation and for providing support for software maintenance. Traceability also provides economic leverage as it details how the system has been developed and avoids the redundant development of certain parts.

To achieve traceability in a software development process, several tasks should be taken into account [2]:

1. Trace definition, to indicate the kinds of objects in our system that can be traced and what information is going to be defined in a trace.
2. Trace production, to indicate what activities, actions, decisions and events happening during software development generate traces for further use.
3. Trace extraction, to indicate how the traces produced in the previous task can be queried in order to achieve certain goals, such as requirements validation or software maintenance.
4. Trace verification, to maintain the integrity of the set of objects and traces.

There are many tools that provide requirements traceability management [11]. To provide efficient traceability management, these tools must resolve certain problems that are present in the industrial setting:

– The lack of a common guideline that describes how to define a traceability model through a well-defined metamodel and how to use it; for example, the way the UML standard provides support to object-oriented modeling. This is due to the variable nature of the traceability capture and use [2], which varies from one organization to another, from one project to another and even from one stakeholder to another.
– The *establish and end-user conflict* [10], where trace providers and users have different goals and priorities.
– The use of heterogeneous tools to define and manipulate the software artifacts involved in a software development process. Thus, the interoperability issue arises as an important feature to be taken into account in a traceability management tool.

These tools are also implemented in an ad-hoc way for the requirements traceability problem without taking into account that the same functionality can be used in other contexts.

3 A Software Maintenance Case Study: Change Propagation

In this case study, we use the change propagate scenario that was introduced in [5]. We illustrate it by means of a specific example shown in Fig. 1. We have defined the information structure of an application in a XML schema (XSD). To build a new

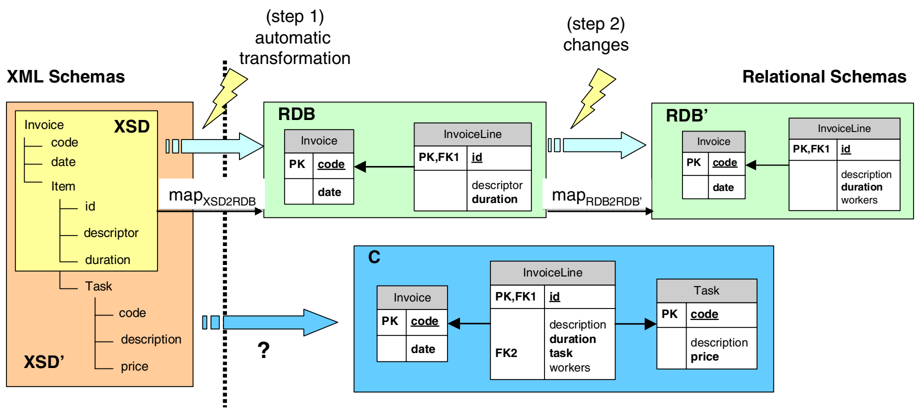


Fig. 1. An example of change propagation

application that stores the information in a relational database, we reuse the metainformation that describes the XML schema. By applying a transformation[1] mechanism (step 1), we obtain the new relational database (RDB). The transformation mechanism also generates a set of links between the new generated RDB relational schema and the source XML schema in order to provide traceability support ($\text{map}_{\text{XSD2RDB}}$).

After obtaining a semantically equivalent relational database from the original XML schema, we continue with the development of the new system. This may involve changes in the application and in the database (step 2), obtaining the relational schema (RDB'). These changes are traced and stored by the tool that manages the model manipulation or by the user directly ($\text{map}_{\text{RDB2RDB'}}$).

Once the new system is developed, changes may occur in the requirements of the system, requiring modifications. It is easier to extend the XML schema than to modify the RDB database. At this point, the application of the transformation mechanism used in step 1 will discard the changes applied from RDB to RDB'.

A solution to this change propagation example can be performed by using model management operators, as shown in Section 6. In our approach, traceability links are used to automate the propagation of changes that were applied to the RDB relational schema, for the new system C.

4 The MOMENT Framework: A MOdel manageMENT Environment into Eclipse

The MDA initiative of the OMG consists of a family of standards that indicate how to define and how to use models to develop software applications in a MDE setting. Application integration and interoperability are two goals of this initiative, as indicated in the request for proposals for the new standard Query/Views/

[1] We use the term 'model transformation' for the mechanism that translates a model between two different metamodels.

Transformations [12]. Nevertheless, to achieve interoperability between applications, bridges built between them are still ad-hoc.

We are working on the application of the model management trend in the context of MDA. We have developed a framework, called MOMENT (MOdel manageMENT), which is embedded in the Eclipse platform. It provides a set of generic operators to deal with models through the Eclipse Modeling Framework (EMF) [13]. EMF provides a close implementation to the MDA guidelines. This framework enables the automatic importation of software artifacts from heterogeneous datasources: UML models (by means of visual modeling environments), relational schemas of any relational database management system (through the Rational Rose tool) and XML schemas. Moreover, third-party researchers and developers are bringing new tools to work on ontologies through EMF [14, 15] and graphical Domain Specific Languages [16, 17]. Therefore, EMF has become an industrial framework for MDA.

4.1 Bridging the EMF and the Maude Technical Spaces

The concept of technical spaces (TS) was introduced by Kurtev et al. in the discussion on the problem of bridging different technologies [18]. A technical space is a working context with a set of concepts, a body of knowledge, tools, required skills, and possibilities [19]. For example, we use the EMF and Maude technical spaces for our framework. The EMF is characterized by its interoperability with industrial tools for solving actual Software Engineering problems. Maude constitutes the formal backbone for our model management approach.

The algebra of operators, which was proposed by Bernstein [20] to deal with models and mappings between models as first-class citizens, has been adapted and directly specified as a generic algebra by using the algebraic specification formalism Maude [21] in the MOMENT framework. This algebraic specification language belongs to the OBJ family, and its equational deduction mechanism animates the specification of an operator over a piece of data, providing the operational semantics for our model management operators. We have developed a plug-in that embeds the Maude environment into the Eclipse framework so that we can use it for our purposes.

In [7, 8], we envisioned the advantages of applying this formalism to solve actual problems in MDA such as model transformation. To fulfill this goal, we have defined two bridges between both technical spaces, at the M2-layer and at the M1-layer (using the Meta-Object Facility [22] terminology). Both of them permit the integration of MOMENT with EMF.

We have defined a projection mechanism at the M2-layer that obtains the algebraic specification[2] that corresponds to a specific metamodel automatically, by applying generative programming techniques. The inverse projection mechanism that obtains an EMF metamodel from an algebraic specification is not interesting in our tool, because the algebraic specification must conform to several features in order to be

[2] The algebraic specification that is generated for a given metamodel (defined in EMF as an Ecore model) permits the representation of models as algebraic terms. Thus, models can be manipulated by our model management operators. Algebraic specifications of this kind do not specify operational semantics for the concepts of the metamodel, they only permit the representation of information for model management issues.

used by our operators, and they should be automatically achieved. We also think that visual modeling environments are more suitable to define such metamodels.

At the model level, we have developed a bidirectional projection mechanism that permits us to project an EMF model as a term of an algebra and to project an EMF model from a term. In this case, the bidirectionality is needed to apply an operator to an input model, since the input model must be serialized as a term and the output term must be deserialized into an EMF model in order to be persisted.

4.2 Operators

In MOMENT, operators are defined in a parameterized module called MOMENT-OP. In this way, operators are defined generically. To apply these operators to specific models, this module must be instantiated by passing a metamodel as actual parameter. This task is automatically performed by the MOMENT tool.

To understand the solution that is given for the change propagation example in Section 6, we informally present some of the model management operators that we use in our approach by indicating their inputs, outputs and semantics:

1. *Cross* and *Merge*
   These operators correspond to well-defined set operations: intersection and disjoint union, respectively. Both operators receive two models (*A* and *B*) as input and produce a third model (*C*). The *Cross* operator returns a model *C* that contains elements that participate in both the *A* and *B* input models; while the *Merge* operator returns a model *C* that contains elements that belong to either the input model *A* or the input model *B*, deleting duplicated elements. Both operators also return two models of links (map_{AC} and map_{BC}) that relate the elements of each input model to the elements of the output model.
   Example: $<C, map_{AC}, map_{BC}> = Cross(A,B)$.
2. *Diff*
   This operator performs the difference between two input models (*A* and *B*). The difference between the two models (*C*) is the set of objects in model *A* that does not correspond to any element in model *B*.
3. *ModelGen*
   ModelGen performs the translation of a model *A*, which conforms to a source metamodel *MMA*, into a target metamodel *MMB*, obtaining model *B*. This transformation implies dealing with two metamodels. This is perfectly feasible in our approach due to the modularity and reusability that algebraic specifications provide. This operator also produces a model of links (map_{AB}) relating the elements of the input model to the elements of the generated model.
   Example: $<B, map_{AB}> = ModelGen_{MMA2MMB}(A)$.

5 Traceability Support in Model Management

All the definitions of Requirements Traceability stated in Section 2 have one feature in common: a trace provides information about a task that has been performed on a source software artifact in a software development process, and relates it to the resulting software artifact. Traceability support must provide both the mechanism that is needed to define traceability links and the query functionality that permits link navigation. In this section, we define traceability in MDA through a model management lens, and we present a set of operators that provide traceability support in the MOMENT framework.

5.1 Generic Traceability Management

A MDA process consists of a sequence of operations performed over a set of models. These models conform to a metamodel and represent specific software artifacts. Operations such as model integration or model transformation can be directly supported by simple model management operators (Section 4.2). Other operations, such as the change propagation mechanism of the case study, can be specified as a complex operator made up of other operators.

Each simple operator carries out a manipulation over a set of input models. To fulfill this, the operator invokes a function that is defined at the metamodel level. The semantics of this function is defined axiomatically in equational logic, and each one of its axioms is called a manipulation rule. To register the task performed over a model, each operator automatically produces a set of links between the elements of a source model and the elements of the resulting model. Such links are stored as models and are used to provide support for traceability.

Following the model management approach, we define Generic Traceability Management as two main issues:

1. The definition of a traceability metamodel to indicate the information needed to link the elements of two different models that can belong to different metamodels in a specific context. The detail of the metamodel depends on the common understanding of the traceability management in a specific society. For example, a generic traceability metamodel may be described for the Requirements Engineering field, although it seems more feasible to define a traceability metamodel for each organization or even each project.
2. A mechanism to extract information from a traceability model independently of the metamodel used. This mechanism is made up of two kinds of operators:
   - Query operators that provide forward and backward navigation through a traceability model.
   - Traceability management operators to manipulate the traceability models in order to automate the reasoning over traceability links. For instance, the *Compose* operator permits chaining traceability links in order to make implicit traceability links explicit; and the *Match* operator permits the inference of traceability models between two models. Furthermore, a traceability model can also be manipulated by model management operators.

5.1.1 Definition of the Traceability Metamodel

To define the traceability metamodel, the user can use the UML notation. This work is done by the user for a specific working context. For the case study, we have specified a traceability metamodel which basically provides the constructs needed to relate elements of a domain model to elements of a target model, independently of their metamodels. In Fig. 2, we show the part of the MOMENT framework metamodel that concerns the traceability support.

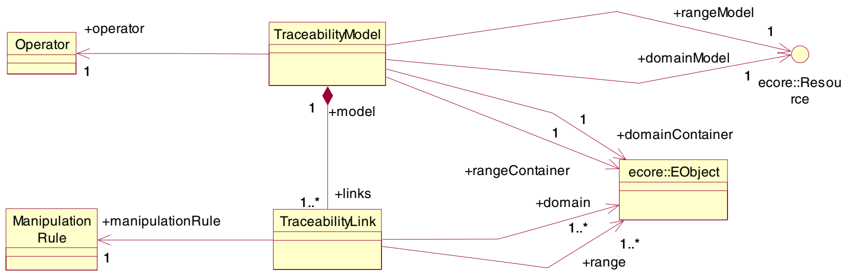


Fig. 2. The default traceability metamodel of the MOMENT framework

In our specific metamodel, the *TraceabilityModel* class is the root element of the package. It allows us to define traceability models. An instance of the *TraceabilityModel* class contains: information about the storage of both the domain and the range model (represented by the interface *ecore::Resource*); which element of each model is the root (by means of the *domainContainer* and *rangeContainer* roles); the operator that has been applied to the domain model; and the links that constitute the traceability model.

The *TraceabilityLink* class indicates how to define a relationship between a set of elements of the domain model and a set of elements of the range model (by means of the *domain* and *range* roles). Each link is associated to the step of the model manipulation task that has produced it (through the *manipulationRule* role). In the metamodel of the figure, *ecore::Resource* and *ecore::EObject* refer to the interface *org.eclipse.emf.ecore.resource.Resource* and to the class *org.eclipse.emf.ecore.EObject*, respectively. The former permits access to a model stored physically. The latter permits access to any element of an EMF model so that any model defined by means of the EMF can be dealt with. The *Operator* class represents an operator that is defined algebraically in MOMENT. The *ManipulationRule* class defines the information needed to specify an axiom for the manipulation function used by the simple operators.

By applying the projection mechanism defined between the EMF TS and Maude TS, we obtain the algebraic specification of the traceability metamodel. This means that we can specify traceability models as sets of elements so that MOMENT operators can be used to manipulate them (*Merge*, *Cross*, *ModelGen*, …).

5.2 Traceability Operators

Once we have shown the definition of a specific traceability metamodel, we explain the generic traceability operators that are provided by MOMENT. The operators that provide support for traceability are defined generically in a parameterized algebraic specification, called *MOMENT-TRAC(Y :: BASICTMM)*.

Fig. 3 shows the elements involved in the parameter passing mechanism diagram. *BASICTMM* (BASIC Traceability MetaModel) is the algebraic specification of the formal parameter, called theory in Maude. This theory declares some operators that guarantee the independence between the semantics of the generic traceability

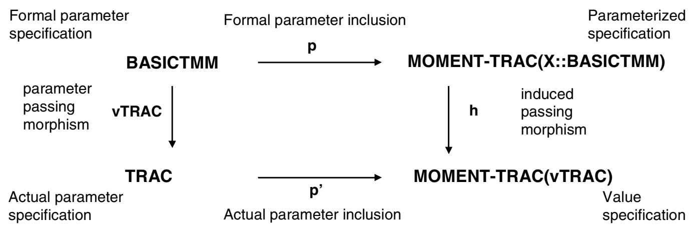

Formal parameter specification
Formal parameter inclusion
Parameterized specification
BASICTMM
p
MOMENT-TRAC(X::BASICTMM)
parameter passing morphism
vTRAC
h
induced passing morphism
TRAC
p'
MOMENT-TRAC(vTRAC)
Actual parameter specification
Actual parameter inclusion
Value specification

Fig. 3. The parameter passing diagram for the *MOMENT-TRAC(Y :: BASICTMM)* parameterized module

operators and the semantics of a specific metamodel. For example, an operator of this kind is *GetDomain*. It obtains the domain element of a traceability link independently of the syntactical representation of the link. Thus, the formal parameter behaves as an interface through which the generic operators can access the elements of a model that conforms to a specific traceability metamodel.

TRAC is the algebraic specification obtained by the projection mechanism from a specific traceability metamodel. The *TRAC* specification constitutes the actual parameter for the *MOMENT-TRAC(Y :: BASICTMM)* module and defines the semantics of the operators that are only declared in the *BASICTMM* theory. The *vTRAC* view is the morphism that relates the elements of the *BASICTMM* formal parameter to the elements of the *TRAC* actual parameter.

The *MOMENT-TRAC(Y::BASICTMM)* parameterized algebraic specification contains the definition of the traceability operators that are independent of the specific TRAC traceability metamodel. The *MOMENT-TRAC(vTRAC)* value specification results from the instantiation[3] of the parameterized module with the specific *TRAC* traceability metamodel.

In this figure, p and p' are inclusion morphisms that indicate that the formal parameter specification is included in the parameterized specification, and that the actual parameter specification is included in the value specification, respectively. The h morphism is the induced passing morphism that relates the elements of the parameterized module to the elements of the *MOMENT(vTRAC)* value specification, by using the *vTRAC* parameter passing morphism.

The traceability operators defined in the *MOMENT-TRAC(X::BASICTMM)* parameterized module are classified in two groups: operators that provide support for navigability and operators that perform tasks on traceability models. In this paper, we focus on the first group of operators.

We define the operators that provide navigability through a traceability model with the following elements: two input models (A and B); a traceability model (map_{AB}) that relates the elements of the two input models and that has been automatically produced by an operator or manually produced by a user; a model (A') that is a sub-model of A (i.e. A' only contains elements that also belong to A); and a model (B') that is a sub-model of B. The traceability operators that are considered here are:

[3] In the context of algebraic specifications, the instantiation of a parameterized module refers to the fact of passing an actual parameter to the parameterized module, obtaining the final value specification.

1. *Domain* and *Range*
   These operators provide the backward and forward navigation through a traceability model, respectively. Both operators obtain a model as output, which is not a traceability model.

   The operator *Domain* takes three models as input: a traceability model (map_{AB}), a domain model (*A*), and a range model (*B'*). The operator navigates the traceability links of the traceability model that have elements of *B'* as target elements, and returns a sub-model of *A* (*A'*), as shown in Fig. 4.a.

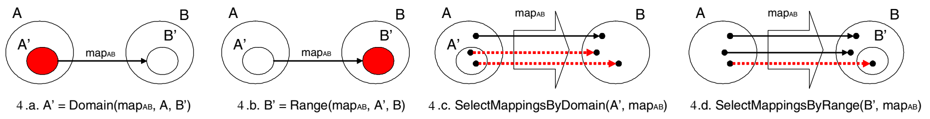

A
B
A'
B'
mapAB
4.a. A' = Domain(mapAB, A, B')
4.b. B' = Range(mapAB, A', B)
4.c. SelectMappingsByDomain(A', mapAB)
4.d. SelectMappingsByRange(B', mapAB)

Fig. 4. Generic operators for traceability navigation

   The operator *Range* also receives three inputs: a traceability model (map_{AB}), a domain model (*A'*), and a range model (*B*). This operator performs the opposite task to the previous one: it navigates the traceability links that have elements of *A'* as domain elements and returns a sub-model of the range model *B* (*B'*), as shown in Fig. 4.b.

2. *SelectMappingsByDomain* and *SelectMappingsByRange*
   These operators produce a traceability model as output and permit selection of parts of a traceability model.

   The operator *SelectMappingsByDomain* receives two input models: a domain model (*A'*) and a traceability model (map_{AB}). The operator extracts the traceability links of the map_{AB} traceability model that have elements of the *A'* model as domain elements and returns this sub-model. The traceability links that are added to the output traceability model are highlighted by a dotted line in Fig. 4.c.

   The operator *SelectMappingsByRange* receives two input models: a range model (*B'*) and a traceability model (map_{AB}). In this case, the operator extracts the traceability links of the map_{AB} traceability model that have elements of the *B'* model as range elements, and returns this sub-model, as shown in Fig. 4.d.

5.3 Process

Taking into account the process described in [2] to define a traceability model, we indicate how its tasks are performed in our tool:

1. Trace definition. Users can define their own metamodel to fit into a specific working context. Moreover the default traceability metamodel of the MOMENT framework can be used in its place. As seen above, the traceability metamodel can be defined using well-known graphical notations, such as UML, through EMF-compliant tools.
2. Trace production. By default, the traceability model is automatically generated by an operator when it performs a manipulation over a set of input models. Nevertheless, traceability links can be defined manually by means of an editor generated from the traceability metamodel, following the EMF software development culture. Moreover,

a traceability model can be inferred automatically between two models by using heuristics [23] or historical knowledge [24].

3. Trace extraction. The analysis of the knowledge provided by a set of traceability models can be useful to perform other tasks. The traceability operators are used to deal with this information in our framework. Such operators constitute an automatic reusable solution that provides support for traceability in many scenarios in the MDE field. Therefore, our framework provides automatic support for this step although the user has to reason about the extracted knowledge. In future works, heuristics may be applied in this step to achieve richer information.
4. Trace verification. The consistency of traceability models can be kept automatically when either their domain or their range model is modified, by means of the application of traceability operators. Consider that we have a domain model *A*, a range model *B* and a traceability model map_{AB} that has been defined between them. Three kinds of model modifications are available: addition of elements, modification of existing elements and deletion of elements. In the case of adding elements to a model, the traceability model remain consistent on the grounds that there is no connection between the new elements and the elements of the other model, unless this connection is defined explicitly afterwards. In the case of modifying and deleting elements of a model, links can be broken when the domain or the range elements are deleted. This problem is easily solved by using traceability operators. If we delete elements in model *A*, obtaining model *A'*, we can apply the *SelectMappingsByDomain* operator to obtain the new consistent traceability model map_{AB}': $map_{AB}' = SelectMappingsByDomain(A', map_{AB})$.

6 Application to the Case Study

The problem explained in the case study can be simplified as shown in Fig. 5, where the $map_{XSD2RDB'}$ traceability model can be easily obtained from the $map_{XSD2RDB}$ and $map_{RDB2RDB'}$ traceability models by means of the *Compose* operator. Therefore, the problem can be enunciated as follows:

We have the following models: an original XML schema (XSD); a XML schema (XSD'), which has been evolved from XSD; a relational database RDB', which has been generated from the XML schema XSD and modified afterwards; and a traceability model between XSD and RDB' ($map_{XSD2RDB'}$). The goal is to obtain a relational database from the XML schema XSD' that preserves the changes applied to RDB'.

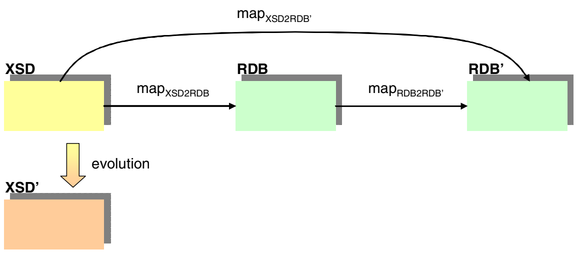


Fig. 5. Schematization of the case study problem

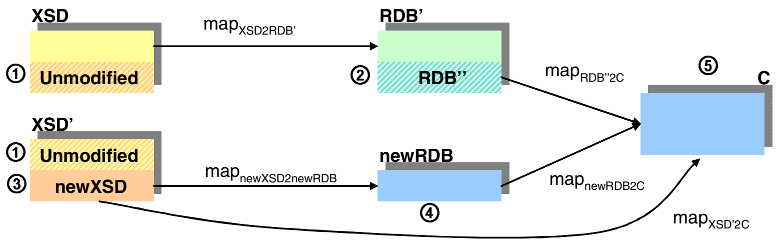


Fig. 6. Solution to the case study problem

This problem can be solved by the following complex operator:

operator PropagateChanges(XSD, XSD', RDB', $map_{XSD2RDB'}$) =	
<Unmodified, $map_{XSD2Unmodified}$, $map_{XSD'2Unmodified}$> = Cross(XSD, XSD')	*(1)*
RDB'' = Range($map_{XSD2RDB'}$, Unmodified, RDB')	*(2)*
<newXSD> = Diff(XSD', Unmodified)	*(3)*
<newRDB, $map_{newXSD2newRDB}$> = $ModelGen_{XSD2RDB}$(newXSD)	*(4)*
<C, $map_{RDB''2C}$, $map_{newRDB2C}$> = Merge(RDB'', newRDB)	*(5)*
return (C)	

This operator is made up of simple operators of the MOMENT algebra and the steps followed in the script are represented in Fig. 6. These steps are the following:

1. *Unmodified* is the part of the *XSD* model that remains unmodified in the *XSD'* model.
2. *RDB''* is the sub-model of *RDB'* that corresponds to the unmodified part of XSD'.
3. *newXSD* is the part of *XSD'* that has been added to the XSD model.
4. *newRDB* is the relational schema obtained from the translation of *newXSD* into the relational metamodel.
5. C is the final model obtained from the integration of the relational databases that we have obtained in steps 2 and 4.

<$map_{XSD'2C}$, $map_{mapUnmodified2C2mapXSD'2C}$, $map_{mapnewXSD2C2mapXSD'2C}$> = Merge(
Compose(Unmodified, SelectMappingsByDomain(Unmodified, $map_{XSD2RDB'}$),RDB'', $map_{RDB''2C}$, C),

If we want to add traceability support to this operator to generate the traceability model that relates the XSD' model to the new model (C) as well, we only have to add the next step after step 5[4]:

This step merges two traceability models: one is defined between the unmodified part of XSD' and C, and the other is defined between the new part of XSD' and C. This step merges both traceability models by means of the *Merge* operator, in the

[4] In this step, the operator Compose (*<map_{AC}> = Compose(A, map_{AB}, B, map_{BC}, C)*) has five input parameters: three models (*A*, *B*, *C*), and two traceability models, which are defined between models *A* and *B* (*map_{AB}*) and between models *B* and *C* (*map_{BC}*). It concatenates both traceability models obtaining a new one that directly relates A to C.

same way that any two models that belong to the same metamodel are merged. The model $map_{XSD'2C}$ has to be added as a return value in the script.

The resulting complex operator solves the change propagation problem of the case study independently of the metamodels involved so that we can apply it to any combination of metamodels, instead of using the XSD and the relational metamodels.

7 Related Work

In the Model Management field, tools do not deal with traceability directly. They usually work on mapping models, which define equivalence relationships between the elements of two models so that a model management operator can be defined generically. Rondo [5] and [6] are good examples of this approach. For instance, Rondo's Merge operator permits the integration of two models. It receives two models (*A* and *B*) and a mapping model (map_{AB}) between them as inputs, and it produces the merged model *C* and two new mapping models (map_{AC} and map_{BC}): $<C, map_{AC}, map_{BC}> = Merge(A, B, mapAB)$.

In MOMENT, mapping models are introduced as traceability models. This is because operators do not have to rely on them to be applied to a set of models. In MOMENT, the traceability relationships between the elements of two models, which are needed to apply an operator to them, are defined between the elements of their corresponding metamodels axiomatically within the corresponding operators. The collection of equivalence relationships between two metamodels constitutes a morphism that can be reused for all the operators of the MOMENT algebra. This permits a clearer specification of complex operators. In MOMENT, the Merge operator is as follows: $<C, map_{AC}, map_{BC}> = Merge(A, B)$. Mapping models are produced by the application of a simple operator to a set of models and keep information about the manipulation task that has been performed to a model. Therefore, we deal with these mapping models from a traceability standpoint.

The Generic Model Weaver AMW [25] is a tool that permits the definition of mapping models (called *weaving models*) between EMF models in the ATLAS Model Management Architecture. AMW provide a basic weaving metamodel that can be extended to permit the definition of complex mappings. These mappings are usually defined by the user, although they may be inferred by means of heuristics, as in [23]. In MOMENT, such mappings are generated by model management operators automatically in a traceability model, and can be manipulated by other operators. We also support extension of the traceability metamodel. Although the simplicity of our initial traceability metamodel, it allows us to deal with complex operators satisfactorily.

8 Conclusions

In this paper, we have presented an overview of our model management approach, focusing on the automatic support that the MOMENT framework provides for traceability. To do this, we have based our approach on the traceability management studies that have been made in the Requirements Engineering field.

We have introduced the Generic Traceability Management concept in the MDA initiative through a Model Management lens. We have also discussed some operators and illustrated how they can be used to solve common software engineering scenarios, like the software maintenance case study presented here. The traceability support has been defined in the MOMENT framework generically so that it can be applied to any context (requirements, workflows, ontologies,…). It can also be customized with a specific traceability metamodel depending on the needs of each working context.

We provide a new vision of the traceability support with respect to previous model management approaches [5, 6], where all operators are based on mappings. In these approaches, the equivalence relationships between elements of two models are defined with specific explicit mappings at the model level. Such mappings are defined by the user or can be inferred by means of heuristics or historical knowledge. However the obtained mappings should be reviewed by a user, which can become burdensome when huge models are involved.

In MOMENT, such equivalence relationships are defined as morphisms between metamodels because algebraic specifications are used as the background formalism. The specification of an equivalence morphism at metamodel level contributes to a more abstract and reusable solution for model management. Users of our model management approach do not have to deal with algebraic specifications directly. The use of EMF to algebraically define models dramatically reduces the learning curve for dealing with models from a formal generic standpoint. It also increases the interoperability with many industrial software development tools.

As we are implementing the MOMENT tool as an Eclipse plugin, AMW constitutes a good environment to define our traceability models by the user. Nevertheless, we are developing our own editor for traceability models in order to add the chance of invoking traceability operators directly from the editor interface. In this way, we will be able to automatically compose traceability models or to navigate them from visual interfaces, using the underlying algebraic specification formalism.

The next step in the MOMENT framework development process is to provide support for the QVT Relations language in order to use it for the definition of equivalence relationships and transformations. The work presented in this paper constitutes the traceability support that MOMENT will provide for the QVT standard.

References

1. Palmer, J.D.: Traceability. Richard H. Thayer and Merlin Dorfman, Software Requirements Engineering, 2nd Edition, IEEE Computer Society, pages 412–422, 2000.
2. Ramesh, B., Jarke, M.: Toward reference models for requirements traceability. Software Engineering, 27(1):58–93, 2001.
3. OMG Model-Driven Architecture. http://www.omg.org/mda/
4. Bernstein, P.A., Levy, A.Y., Pottinger, R.A.: A Vision for Management of Complex Models. Microsoft Research Technical Report MSR-TR-2000-53, June 2000, (short version in SIGMOD Record 29, 4 (Dec. '00)).
5. Melnik, S., E. Rahm, P. A. Bernstein, "Rondo: A Programming Platform for Generic Model Management," Proc. SIGMOD 2003, pp. 193-204 (PDF, 344KB). Extended version in Web Semantics, Volume 1, Number 1.
6. Song, G., Zhang, K., Kong, J.: Model Management Through Graph Transformation. IEEE VL/HCC'04. Rome, Italy. 2004.

7. Boronat, A., Pérez, J., Carsí, J. Á., Ramos, I.: Two experiencies in software dynamics. *Journal of Universal Science Computer*. Special issue on Breakthroughs and Challenges in Software Engineering. Vol. 10 (issue 4). April 2004.
8. Boronat, A., Carsí, J.Á., Ramos, I.: Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine. IEEE Computer Society Press. 9th European Conference on Software Maintenance and Reengineering. Manchester, UK. 2005.
9. M. Dorfman and R. Thayer. Guide to software requirements specification. IEEE Standards, Guidelines and Examples on System and Software Requirements Engineering, 1990.
10. Gotel, O. C. Z. and Finkelstein, A. C. W.: An Analysis of the Requirements Traceability Problem, Proceedings of the First International Conference on Requirements Engineering (ICRE '94), IEEE Computer Society Press, Colorado Springs, Colorado, U.S.A., April 18-22, pp. 94-101.
11. Requirements tools in the Volere web site: http://www.volere.co.uk/tools.htm
12. Object Management Group. Request for Proposal: MOF 2.0 Query/Views/ Transformations RFP, 2002. ad/2002-04-10.
13. The EMF site: http://download.eclipse.org/tools/emf/scripts/home.php
14. Zhang, L., Yu, Y., Lu, J., Lin, C., Tu, K., Guo, M., Zhang, Z., Xie, G., Su, Z., Pan, Y.: ORIENT: Integrate Ontology Engineering into Industry Tooling Environment. In Proc. of the Third International Semantic Web Conference 2004, Hiroshima, Japan.
15. The Hyena web site: http://www.pst.ifi.lmu.de/~rauschma/hyena/
16. The Graphical Modeling Framework proposal: http://www.eclipse.org/proposals/eclipse-gmf/main.html
17. The JANE Model-specific editor generator web site: http://www.dstc.edu.au/Research/ Projects/Pegamento/jane/
18. Kurtev, I., Bézivin, J., Aksit, M.: Technological Spaces: An Initial Appraisal. Int. Federated Conf. (DOA, ODBASE, CoopIS), Industrial track, Irvine, 2002.
19. Bézivin, J., Devedzic, V., Djuric, D., Favreau, J.M., Gasevic, D., Jouault, F.: An M3-Neutral infrastructure for bridging model engineering and ontology engineering. In Proceedings of INTEROP-ESA'05, Geneve, Switzerland. 2005.
20. Bernstein, P.A: Applying Model Management to Classical Meta Data Problems. pp. 209-220, CIDR 2003.
21. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theoretical Computer Science, 285(2):187-243, 2002.
22. OMG, "Meta Object Facility 1.4", http://www.omg.org/technology/documents/formal/ mof.htm
23. Madhavan, J., P.A. Bernstein, and E. Rahm: Generic Schema Matching using Cupid. VLDB 2001.
24. Madhavan, J., Bernstein, P. A., Chen, K., Halevy, A.Y., Shenoy, P.: Corpus-based Schema Matching," Workshop on Information Integration on the Web, at IJCAI'2003, pp. 59-66.
25. Didonet Del Fabro, M, Bézivin, J, Jouault, F, Breton, E, and Gueltas, G : AMW: a generic model weaver. Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05). 2005.

Synchronizing Cardinality-Based Feature Models and Their Specializations

Chang Hwan, Peter Kim, and Krzysztof Czarnecki

University of Waterloo, Canada
{chpkim, kczarnec}@swen.uwaterloo.ca

Abstract. A software product line comprises a set of products implementing different configurations of features. The set of valid feature configurations within a product line can be described by a feature model. In some practical situations, a feature configuration needs to be derived in stages by creating a series of successive specializations of the initial feature model. In this paper, we consider the scenario where changes to the feature model due to, for example, the evolution of the product line, need to be propagated to its existing specializations and configurations. After discussing general dimensions of model synchronization, a solution to synchronizing cardinality-based feature models and their specializations and configurations is presented.

1 Introduction

Feature modeling is a systematic way of describing variabilities and commonalities of systems in a software product line [1,2]. A feature model describes a set of possible configurations or combinations of features. In this paper, we focus on a particular style of feature models referred to as *cardinality-based feature models* [3].

A configuration can be arrived at in stages, where at each stage some choices are made [3]. The outcome of each stage is a feature model which is a specialization of the input feature model for that stage. A specialization of a feature model describes a subset of the configurations represented by that model. The need for staged configuration arises in several practical situations, such as in

– *software supply chains,* e.g., a platform vendor may need to make some configuration choices to a platform before releasing it to a specific customer, and the customer may need to provide further settings for individual applications;
– *optimization,* e.g., certain configuration choices could be made at compile time, while remaining ones are decided at runtime; the application code could be optimized based on the compile-time choices;
– *multi-level policies,* e.g., security policies may be specialized at different levels of an organization.

In any realistic setting, the variabilities and commonalities in a product-line will evolve. Inevitably, a feature model will also have to change, and the existing specializations of multiple stages will have to be synchronized in order to reflect the change in the feature model. The interesting challenge is to perform synchronization with the intent of preserving the choices made in the stages of specializations. For simplicity, we refer to the synchronization of cardinality-based feature models and their specializations and configurations as *feature-model synchronization.*

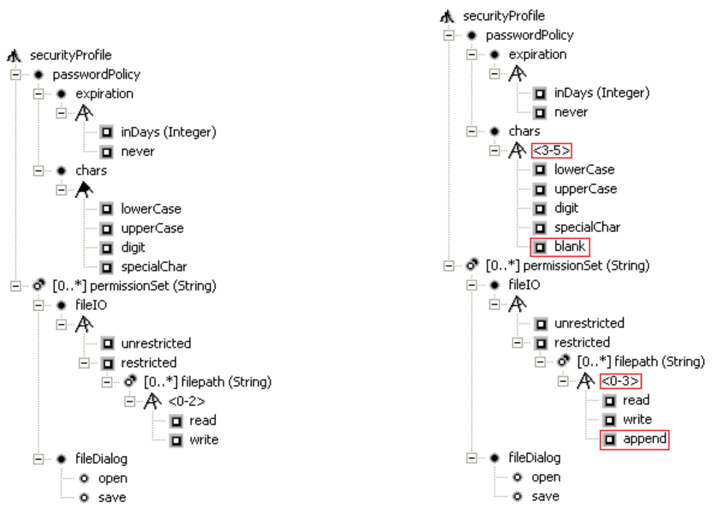
In this paper, we first characterize feature-model synchronization according to dimensions that are applicable to other model synchronization problems. Then we give a solution to the feature-model synchronization problem. The solution and the significant issues surrounding the problem are explained in natural language. The main characteristic of the solution is that it treats feature models, specializations, and configurations in a uniform way. Additionally, we describe how the solution can be specified using the *Relations* language from the latest submission for the Object Management Group's MOF 2.0 Query/View/ Transformation standard [4]. To our knowledge, feature model synchronization in a multi-staged configuration setting is a problem unexplored until now. We believe that the results we present are novel contributions with relevance to both the software product-line community and the model-based development community.

In Section 2, the synchronization problem is motivated through an example. In Section 3, general dimensions of model synchronization are discussed. In Section 4, feature model synchronization is characterized and a technique to achieve it is described. Section 5 discusses related work. Section 6 concludes the paper.

2 Background and Motivating Example

A *feature model* is a hierarchy of features plus constraints describing valid *configurations* of features. Features are used to model functional and non-functional characteristics of systems, but for the purpose of our discussion, they are just symbols with no further semantics. A feature model always has a *root feature.* The remaining features are either *grouped* or *solitary*, i.e., they are either part of a *feature group* or not. Each solitary feature is annotated with a *feature cardinality*, which is an interval constraining how many times the feature has to be included in a configuration if its parent is also included.[1] Each feature group is annotated with a *group cardinality*, which is an interval constraining how many features from that group have to be included in a configuration if the parent feature of the group is also included. Additional constraints on possible configurations may also need to be expressed, such as *requires* and *excludes* constraints. Constraints may be specified using XPath, as explained elsewhere [5]. A feature may be associated with an attribute type, in which case an attribute value can be specified during configuration.

[1] As explained later in Section 4.2, we also associate feature cardinalities with grouped features. However, the only possible values are [0..0], [0..1], and [1..1]. Normally, a grouped feature has [0..1] as its default cardinality. The cardinalities [1..1] and [0..1] are used for features that were selected or eliminated, respectively, from a group during specialization.



(a) Initial feature model (b) Modified feature model

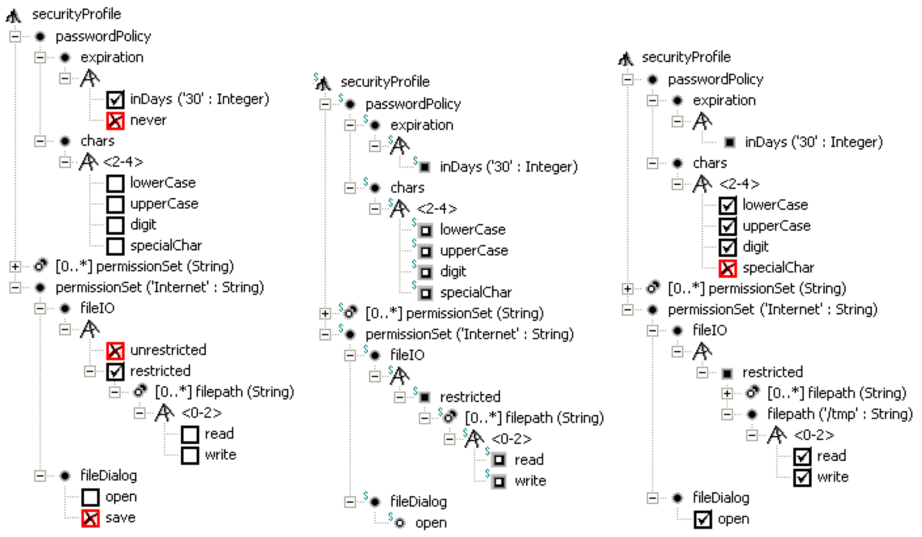
Fig. 1. Security profile feature model before and after changes

Table 1. Symbols used in cardinality-based feature modeling

Symbol	**Explanation**
⋔	Root feature
●	Solitary feature with cardinality [1..1], i.e., *mandatory* feature
○	Solitary feature with cardinality [0..1], i.e., *optional* feature
○ $[0..m]$	Solitary feature with cardinality $[0..m]$, $m > 1$, i.e., *optional clonable* feature
● $[n..m]$	Solitary feature with cardinality $[n..m]$, $n > 0 \wedge m > 1$, i.e., *mandatory clonable* feature
◘	Grouped feature
$f(\text{'}value\text{'} : T)$	Feature f with attribute of type T and value of $\text{'}value\text{'}$
△	Feature group with cardinality $\langle 1\text{–}1 \rangle$, i.e. *xor*-group
▲	Feature group with cardinality $\langle 1\text{–}k \rangle$, where k is the group size, i.e. *or*-group
△ $\langle i\text{–}k \rangle$	Feature group with cardinality $\langle i\text{–}k \rangle$

Figure 1(a) shows a sample feature model describing the configuration choices available in a security profile of an operating system. The notation is explained in Table 1. The profile contains a policy for password expiration, which can be never or after a specific number of days, and for the kind of characters required in a password. The permission set specifies allowable operations on various resources such as files, file dialogs, and environment variables. A configuration can have multiple permission sets, e.g., a permission set defined for code downloaded from the Internet and another for code coming from the intranet of an organization.

A *specialization* of a feature model is another feature model which describes a subset of the configurations represented by the original feature model.



(a) Specialization of 1(a) in check-box view (b) Specialization of 1(a) in feature model view (c) Configuration of 2(b) in check-box view

Fig. 2. Specializations of the initial feature model from Figure 1(a)

A specialization can be created by first copying the original model and then applying *specialization steps* to the copy [6]: select or eliminate an optional solitary feature, select or eliminate a grouped feature from a group, refine feature cardinality, refine group cardinality, clone a clonable solitary feature, and assign value to a feature attribute.[2] Figure 2(a) shows a specialization of the feature model from Figure 1(a) in the *check-box view*, which supports the application of the specializations steps. Check boxes are shown for optional solitary features and grouped features. Placing a check on a check box corresponds to selecting a feature. A cross corresponds to eliminating the feature. An empty check box means no change. Cardinalities can be refined by editing them. The clone operation can be invoked on clonable features through the context menu. Values can be assigned to feature attributes. The resulting specialization rendered in the *feature model view* is shown in Figure 2(b). Note that we use a filled square to indicate a selected grouped feature, e.g., *inDays* and *restricted*. Compared to the original model, in the specializations shown in the check-box view in 2(a) and in the feature model view in 2(b), `never` is eliminated, `inDays` is assigned an attribute value of 30, the cardinality of the group under `chars` is refined to $\langle 2\text{–}4 \rangle$, `permissionSet` has a clone with the attribute value `Internet` assigned to it. Furthermore, `restricted` is selected in the clone, and `unrestricted` and `save`

[2] The original description of specialization steps [6] includes unfolding feature model references. We do not consider the latter in this paper for simplicity. Furthermore, selecting or eliminating an optional solitary feature is equivalent to refining its cardinality to [1..1] or [0..0], respectively.

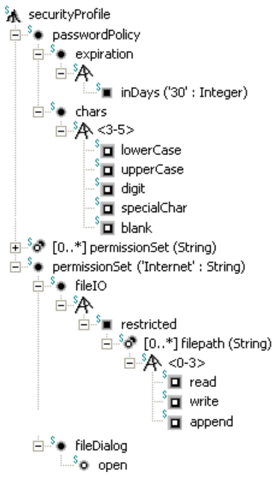
are eliminated. The *prototype* feature for the clone, i.e., `permissionSet` with cardinality [0..∗], is shown in collapsed form for brevity.[3] An alternative rendering could suppress the display of the groups under `expiration` and `fileIO` and show *inDays* and *restricted* as mandatory subfeatures of `expiration` and `fileIO`, respectively.

The specialization in Figure 2(b) could have been created by an organization to take their special security requirements into account. Further specializations could be created based on the presented one, for example, in order to satisfy the security constraints of individual departments. Finally, the department-level specializations could be used as a basis for creating configurations for individual computers.

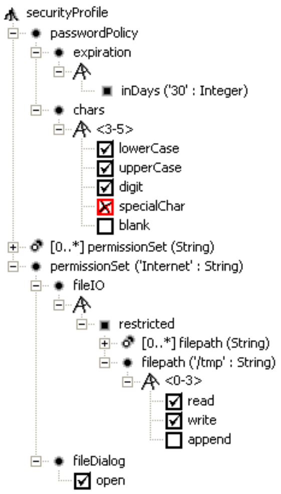
The relationship between a feature model and its *configuration* corresponds to the one between a type and its instance. A feature model in which there is no variability represents exactly one configuration. Such a feature model, which is comparable to the notion of a singleton type, can also be used in place of the configuration it describes. This observation allows us to treat configurations as feature models, too. As a consequence, a tool can use the same interface for creating specializations and configurations, e.g., the check-box view, and configurations and specializations can be also rendered in the feature-model view. Figure 2(c) shows a sample configuration, which was created based on the specialized feature model from Figure 2(b). Note some of the choices made, including the elimination of `specialChar` and the selection of `open`. Figure 2(c) shows a configuration even though it still contains `filepath` with cardinality [0..∗]. This is because we assume that features with cardinality $[0..n]$, where $n > 1$, are by default not part of the configuration. Similarly, undecided optional features, i.e., those with empty check boxes, could also be considered as by default not part of the configuration. We keep features such as `filepath` with cardinality [0..∗] in the check-box view as prototypes, should the user decide to create more clones. If desired, the prototype `filepath` can be eliminated explicitly by refining its cardinality to [0..0].

Any changes made to a feature model need to be propagated to all its specializations and configurations, which is an example of model synchronization. For example, the available security settings described by the feature model in Figure 1(a) could change in the next release of the operating system to those described by the feature model in Figure 1(b) (the changes are highlighted). Comparing the models reveals that `blank` character type has been added, the cardinality of the group containing `blank` has been changed to $\langle 3\text{–}5 \rangle$, `append` has been added to the group under `filepath`, and that group's cardinality has changed to $\langle 0\text{–}3 \rangle$. All these changes need to be propagated to the specialization and configuration in Figure 2, whose updated versions are shown in Figure 3. Please note that the configuration from 2(c) became a specialization in Figure 3(b) due to the newly added features `blank` and `append`, which are undecided. Furthermore, the organization could decide to change the specialization in Figure 2(a) by applying further specialization steps or undoing some of the previously applied ones. These changes would also need to be propagated to the configuration in Figure 2(c).

[3] The feature from which a clone is created is referred to as the *prototype* of the clone.



(a) Specialization shown in 2(a) and 2(b) updated



(b) The resulting specialization after updating the configuration in 2(c)

Fig. 3. Specializations updated according to the changed feature model from Figure 1(b)

3 General Dimensions of Model Synchronization

In this section, we analyze dimensions of model synchronization in general and classify feature model synchronization in terms of these dimensions. This exercise will help us to better understand the characteristics of feature model synchronization and devise a solution in Section 4.

Model synchronization is concerned with maintaining consistency among two or more models in the presence of changes to one or more of them. Synchronization includes both the detection of inconsistencies among the models and modification of one or more models in order to re-establish consistency. The modification is referred to as *reconciliation*.[4]

Structural characteristics. Synchronization problems can be characterized through their structural properties:

– **Number of models and synchronization direction.** Synchronization can be thought of as a procedure with two or more input parameters, some or all of which are also output parameters. Even more generally, different subsets of the input parameters could be additionally marked as output parameters for different individual invocations of synchronization. A common case is *unidirectional* synchronization between two models, where a

[4] There is no fundamental difference between synchronizing a set of models and synchronizing different parts of one model because the set of models could be viewed as parts of one large model. However, in order to keep our discussion more clear, we take the former rather than the latter view.

change to a source model needs to be propagated to a target model. Unidirectional synchronization corresponds to making the source immutable and target mutable for the synchronization procedure. In contrast, *bidirectional* synchronization between two models treats both models as mutable.
- **Types of models.** Each model involved in synchronization conforms to a metamodel. Each model could conform to a different metamodel, or some or all of the models could conform to the same metamodel.
- **Multiplicity of relationships among elements.** The consistency among the involved models can be expressed as relationships among their elements. The instances of these relationships are traceability links. An important characteristic of the relationships is their multiplicities, such as *one-to-one*, *one-to-many*, and *many-to-many.* Models involving one-to-one or one-to-many relationships are usually easier to synchronize than those involving many-to-many relationships.

User-facing characteristics. Several characteristics are related to how synchronization is triggered and performed from the user viewpoint:

- **Point in time and frequency of triggering.** Synchronization could be performed *continuously*, where every change to a model is immediately propagated to the other models, or could be triggered *on demand.* The continuous mode is inappropriate in most practical cases. The continuous mode would slow down editing because every editing operation would trigger change propagation. Also, we usually do not want to keep models synchronized at all times; in particular, we do not want to synchronize with all intermediate states of a model while the model is edited. A more practical choice is to allow the user to invoke synchronization when he deems it appropriate.
- **Scope of synchronization.** Synchronization can be performed in a *push* or *pull* mode. In the push mode, a change to a model is pushed to all dependent models. In the pull mode, each dependent model can be synchronized individually on demand. An important consideration is whether all the models to be synchronized are available within a single development environment, or whether they are distributed among different physical locations. In the latter case, the push mode would usually involve pushing change notifications only, while the user at the receiving side would decide whether and when to perform synchronization.
- **Reconciliation strategy.** In general, different reconciliation strategies may involve different degrees of *automation* versus *interaction with the user*, *preservation of the existing model structure*, and *completeness.* During reconciliation, some model changes may be determined as necessary. They can be performed automatically. However, different alternative changes may be available in order to achieve further progress with reconciliation. The selection of alternative changes could be automated by making some default choices, or the alternatives could be presented to the user so that choices can be made interactively. Furthermore, the different choices may involve different degrees of modification to the existing elements of the models being

reconciled. If the model to be changed by reconciliation has been automatically generated, overriding existing structures may not be an issue; however, if the model contains manual modifications, they should be preserved as much as possible. Finally, the reconciliation process could be partial. The remaining inconsistencies would be marked as such and left to the user to be resolved manually.

Implementation Choices. Two main issues need to be considered when implementing model synchronization:

- **Representation of the synchronization logic.** The synchronization logic could be expressed in *action-* or *state-oriented* style. In the action-oriented style, edit operations on one model are mapped to edit operations on the dependent models. Such a mapping could be used to implement the continuous mode of synchronization. On-demand mode could be achieved by mapping whole histories of operations. The history of the source model could be recorded during editing, or it could be created by comparing the version of the model before the edits with the version after the edits. In the state-oriented style, the models to be synchronized are analyzed, and one or more of them are transformed to re-establish consistency. The state-oriented approach may sometimes require access to the source model versions before and after editing in order to properly propagate changes. The necessary model transformation can be expressed algorithmically or as a set of transformation rules. The transformation rules can address synchronization explicitly. Alternatively, synchronization can be defined implicitly by model consistency rules. In the former case, the transformation rules will read source and target models and modify the target models to establish consistency. In the latter case, the consistency between the models is specified as rules, and a rule engine will attempt to fix any violations of these rules by modifying the target models. The engine may need to interact with the user, apply some heuristics, and use additional problem-specific strategies to perform the reconciliation.
- **Representation of traceability links.** Traceability links can be explicit or implicit in the synchronization rules. Implicit links are established as instances of pattern matching. They are created and maintained by the rule engine. Explicit links need to be taken care of explicitly by the user. They are not automatically created and maintained by the rule engine. Traceability links can be implemented using in-memory pointers and/or globally unique identifiers of model elements.

4 Feature Model Synchronization

In the following three subsections, we first characterize feature model synchronization using the dimensions introduced in Section 3, then specify the metamodel for representing feature models, and, finally, describe how feature model synchronization can be performed.

4.1 Classification of Feature Model Synchronization

Feature model synchronization can be characterized in terms of the general dimensions for model synchronization as follows:

Structural characteristics. Assuming that specializations and configurations are feature models (as explained in Section 2), feature model synchronization is a unidirectional synchronization between two feature models. In other words, both models conform to the same metamodel. Furthermore, because of cloning, the relationship between elements in a feature model and the elements in one of the feature model's specializations or configurations, is in general, one to many rather than one to one.

User-facing characteristics. Feature model synchronization requires activation on demand. Both pull and push modes are of interest. We also need to consider both synchronization among feature models loaded within one development environment and those distributed to different users. Preservation of existing user choices in specializations and configurations is absolutely important. Therefore, the synchronization process will usually be only partially automatic. Remaining inconsistencies, such as violated cardinality or additional constraints, will be marked. The user can resolve them by using the constraint-solving facilities normally available in feature model specialization and configuration [7]. In other words, an automatic synchronization phase is followed by a tool-supported, interactive phase, in which alternative conflict resolution choices and completions based on defaults are presented to the user.

Implementation Choices. Feature model synchronization can be implemented using the variety of implementation choices listed in Section 3. In the further discussion, we only consider the state-oriented approach. Since continuous update is inappropriate for feature model synchronization, the action-based approach would need to operate on histories. We find the state-oriented approach simpler and more robust since it does not need to consider histories. In Section 4.3 and Appendix A, we will discuss an algorithmic solution and a rule-based one using consistency rules, respectively. Only the latest versions of the source and target models are needed for feature model synchronization. The presented solutions use explicit links, which are established automatically when a copy of a feature model to be specialized is first created. In the case of feature model synchronization, the additional complexity of maintaining the links by the synchronization algorithm explicitly is minimal.

4.2 Metamodel and Renderings for Cardinality-Based Feature Models and Their Specializations

Before getting into the specifics of synchronization, the underlying representation of cardinality-based feature models and their specializations needs to be understood. Figure 4 shows the metamodel for representing feature models and their specializations and configurations. A feature model is a hierarchy of `Nodes` modeled by the `parent-child` composition. A `RootFeature` is the only `Node` without a `parent`. A `Feature` or a `RootFeature` may have `Features`

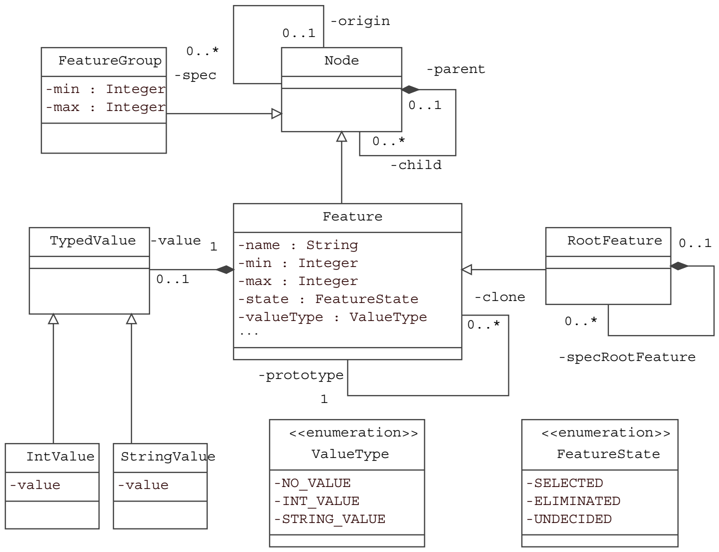


Fig. 4. Metamodel for cardinality-based feature models and their specializations

and/or `FeatureGroups` as children. Each child of a `FeatureGroup` must be a `Feature`. `Features` and `FeatureGroups` use `min` and `max` to represent cardinalities. The cardinality of root and grouped features is always $\langle 1-1 \rangle$. A `Feature` may have an attribute, in which case its type is indicated by `valueType`. A `Feature` with an attribute may contain a `TypedValue`. The `specRootFeature` composition on `RootFeature` is used to represent a hierarchy of specializations. The `RootFeature` of a feature model contains the `RootFeatures` of its direct specializations and configurations, which are feature models too. Traceability links between `Nodes` of a feature model and its specialization are modeled by the `origin-spec` association. Group and feature cardinalities in the nodes of a specialization are used to override cardinalities of their `origin` nodes. A specialization cannot add an attribute, but only an attribute value. Adding an attribute value is only possible if a value has not been yet assigned in the previous stage. `Feature` has the field `state`, which is used to represent specialization choices on optional solitary features and grouped features, which are made in the check-box view, as shown in Figure 2(a). The allowed values for `state` are `SELECTED`, `UNDECIDED`, or `ELIMINATED`. The meaning of these values will be explained shortly. The relationship `clone-prototype` is used to relate clones to its prototype feature, i.e., the feature from which they were cloned.

When a feature model is first created in the feature model editor as in Figure 1(a), none of the nodes in the model have a corresponding `origin` node. A specialization of the feature model is created by first creating a copy of it and then setting the origin of each node in the specialization to the corresponding node in the original feature model. The feature model editor showing the original feature model renders the model according to Table 1, where the group and feature

Table 2. Interpretation and rendering of `min`, `max`, and `state`

Stored cardinality [`min`..`max`]	**Value of state**	**Effective cardinality**	**Rendering in check-box view**	**Rendering in feature-model view**
$\texttt{min}-c \leq 0 \wedge$ $\texttt{max}-c \leq 0$	—	[0..0]	✗ or feature not shown	
$\texttt{min}-c \leq 0 \wedge$ $\texttt{max}-c = 1$	`ELIMINATED`	[0..0]	☒	✗ or feature not shown
	`UNDECIDED`	[0..1]	☐	○ for solitary and ▫ for grouped feature
	`SELECTED`	[1..1]	☑	● for solitary and ■ for grouped feature
$\texttt{min}-c = 1 \wedge$ $\texttt{max}-c = 1$	—	[1..1]	● for solitary and ■ for grouped feature	
$\texttt{min}-c \leq 0 \wedge$ $\texttt{max}-c > 1$	—	[0..$\texttt{max}-c$]	○ [0..$\texttt{max}-c$]	
$\texttt{min}-c \geq 1 \wedge$ $\texttt{max}-c > 1$	—	[$\texttt{min}-c$..$\texttt{max}-c$]	● [$\texttt{min}-c$..$\texttt{max}-c$]	

cardinalities shown in this rendering are those *stored* in the model instance, i.e., [`min`..`max`] and ⟨`min`−`max`⟩, respectively. Rendering of specializations is different. First, specializations are not to be shown using the feature model editor, as free editing is not allowed for a specialization. Instead, a specialization can be shown in (1) the check-box view (e.g., Figure 2(a)), which allows the application of specialization steps, or (2) the feature model view (e.g., Figure 2(b)), which renders the result of the specialization as a feature model and does not allow any editing. Second, the views do not display the cardinalities stored in the specialization instance, i.e., [`min`..`max`] and ⟨`min`−`max`⟩, directly. Instead, both views render the model nodes according to their *effective cardinality*. The effective cardinality for a feature and its corresponding rendering in the check-box and feature model views are given in Table 2. In that table, c refers to the number of clones of that feature. An entry of "—" in the second column means that the value of `state` is insignificant for the case represented by the corresponding row. Feature groups are rendered according to Table 1 and their effective cardinality. The effective cardinality of a feature group with the group cardinality ⟨`min`−`max`⟩ and k features is defined as $\langle \texttt{min}-min(\texttt{max}, k-e)\rangle$, where e is the number of features with the effective feature cardinality of [0..0] that are contained in the group. The specialization step of feature or group cardinality refinement can be achieved in the check-box view by directly editing the cardinality stored in a node, with the only constraint that the refined *stored* cardinality is a subinterval of the *effective* cardinality of the node's `origin`.

4.3 Feature Model Synchronization Steps

Synchronization between a feature model and a specialization starts with an automatic phase, possibly followed by an interactive phase. During the the automatic phase, synchronization steps are applied to the specialization, such as adding, deleting, and relocating nodes, adding and removing attributes, and adjusting attribute values and cardinalities. The interactive phase may be necessary in order to enforce new cardinality values by deleting or creating clones and changing feature selections within groups and in order to re-establish global constraints (such as implies and requires constraints) by further reconfiguration. The changes in the interactive phase cannot be fully automated because there may be many different ways to enforce cardinalities and global constraints and the user may need to be consulted.

In the case of multiple specializations in a multi-stage scenario, the synchronization steps need to be repeated for each stage. The direct specializations of the feature model that was modified are synchronized first. Then the direct specializations of the newly synchronized specializations are synchronized. This process continues recursively until all models are synchronized.

The automatic synchronization steps are described in the remainder of this section in an algorithmic style. In Appendix A, we indicate how these steps can be represented as consistency rules between a model and its specializations in the OMG QVT *Relations* language [4].

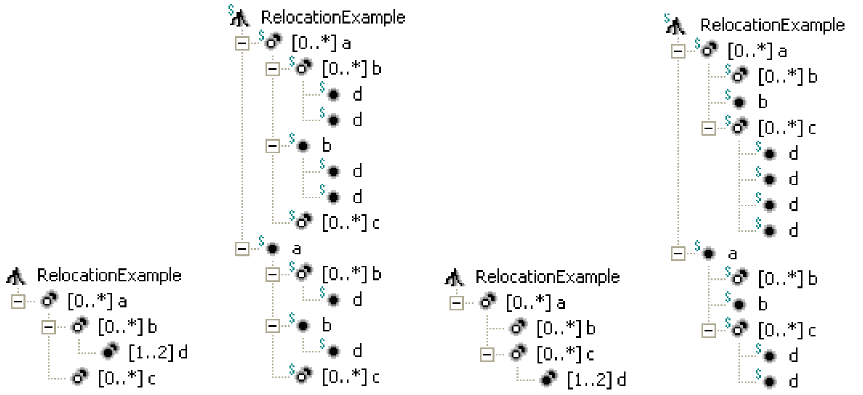
Addition. Node instances that are missing in the specialization are added. For every node `m` in the original model and for every node `s` in `m.spec`, if `m` has a child `mChild` for which there is no child `sChild` of `s` such that `sChild.origin` = `mChild`, such an `sChild` is created. Note that since all the cloned features in the specialization have a traceability link back to the original-model feature, adding nodes underneath that feature means that the specializations of the nodes will be added to each clone.

Removal. Node instances in the specialization whose `origin` attribute is null are removed.

Changing attribute. If a feature `f` in the original model has an attribute type, i.e., `valueType` $\neq$ `NO_VALUE`, every feature in `f.spec` must have the same attribute type. Furthermore, if `f` has an attribute value, every feature in `f.spec` must also have the same attribute value.

Relocation. A model node and everything below may be moved from one parent to another parent. As a result, the specializations of the changed model are out of sync with the model. In particular, the parent of the model node and the parent of a corresponding specialization node will not be connected by the `origin-spec` traceability link. Due to the possible presence of clones, relocating out-of-place specialization nodes is a challenge for synchronizing cardinality-based feature models. Consider a simple feature model in Figure 5(a) and its specialization in 5(b). Imagine that `d` is moved under `c`, as shown in Figure 5(c). There are different ways to synchronize. One way is simply to take all the specialization nodes of `d` and copy them for each specialization node of `c`. There are six `d` specialization nodes in 5(b). That means there would be six `d` specialization nodes in each of the two `c` specialization nodes. In general, because clones are collected throughout the hierarchy, this solution tends to produce an excessive number of clones. Also, the high number of clones will often lead to cardinality violations. The model allows a maximum of two clones of `d`, but the relocation, as described, yields six clones.

Another way is to perform relocation under a well-defined scope, as shown in Figure 5(d). We could take the source container node, or `b`, in this case, and the target container node, or `c`, and find the common ancestor node between the two in the model, which is `a`. We could perform the relocation under the scope of the common ancestor, by taking all the nodes of `d` under the ancestor and copying them under each target node of `c` under the ancestor. In this case, four `d` nodes would end up under the first `c` node and two `d` nodes would end up under the second `c` node. Still, there is a cardinality violation, as four `d` clones are not allowed according to the feature model. However, the move performed against the second `c` clone is fine. This relocation method attempts to reduce the probability and extent of cardinality violation. Nodes violating cardinality are marked as such to allow the user to fix them manually.



(a) Model (b) Specialization (c) Modified model (d) Updated specialization

Fig. 5. Feature relocation example

The synchronization mechanism described can also be used for moving a solitary feature into a feature group in the model. Moving the solitary feature into the feature group this way would mean that both the corresponding prototypes and their clones in the specialization would all end up in the corresponding feature group. Alternatively, only the prototypes can be moved into the feature group, discarding the clones. In any case, the prototypes and clones will be marked for the user to be inspected manually. The user will have to choose exactly one prototype or clone to keep among the prototypes and clones with the same origin and delete the remaining ones.

Cardinality changes. The stored cardinality of every feature group and feature prototype in a specialization has to be a subinterval of the effective cardinality of the corresponding origin group or feature. If this is not the case, `min` and/or `max` of the feature group or feature prototype need to be adjusted to enforce this constraint. Finally, the number of clones in the specialization may violate the stored cardinality of their corresponding prototype feature. Also, the number of selected features in a group may violate the stored group cardinality Both kinds of violations have to be resolved by the user, as they may involve deciding which nodes to delete and which nodes to create.

5 Related Work

We are not aware of any previous work on feature model synchronization. The closest bodies of work are that on model synchronization and schema evolution.

There has been a considerable amount of effort in the model-driven development community to provide generic frameworks for model transformation that support synchronization. An OMG standard for model transformations called "MOF 2.0 Query/View/ Transformation (QVT)" [4] is under development. In Appendix A, QVT *Relations* language is demonstrated through the specification of synchronization rules. Although there does not yet exist a publicly available implementation of QVT, a prototype implementing some of its ideas was developed by IBM under the name *Model Transformation Framework (MTF)* [8]. MTF provides a concise language for specifying equivalence relations for models represented using the Eclipse Modeling Framework (EMF) and a transformation engine. We have experimented with MTF as an infrastructure to implement feature model synchronization for FeaturePlugin [5], a feature modeling tool for Eclipse. The prototype implementation was able to synchronize node additions and removal. Unfortunately, in its current state, MTF simply lacks support for defining the custom constraints that are required to fully achieve synchronization. A comparable approach to MTF is *ModelWeaver*, as proposed by Bezivin et al. [9]. However, in contrast to MTF, which is EMF-centric, the ModelWeaver approach focuses on relationships between different technical spaces, such as MOF, XML, EMF, GrammarWare, etc.

Other works in the model transformation area include the efforts of Ivkovic and Kontogiannis on synchronizing software artifacts across levels of abstraction [10]. In their approach, model dependencies are implicitly encoded using transformation rules and an equivalence relation is used to evaluate when two models become synchronized. Furthermore, Mens et al. use description logics to synchronize between UML models [11].

The synchronization problem has also been explored in the context of schema evolution, for example, in object-oriented databases [12,13,14]. Bernstein et al. describe a general model management framework for schema evolution in which mappings between models are treated as first class objects and operators are defined for common operations on models, such as merging, matching, and differencing [15]. They provide a concrete implementation of the framework called *Rondo* that implements some model operators [16]. Also, Sprinkle et al. describe the use of a graphical model transformation language to migrate models form one version of a metamodel to another version [17]. The relationship between a feature model and its configurations is comparable to that of a schema and the data conforming to that schema. However, in contrast to schema evolution, feature model evolution considers multiple stages of specialization.

Mens et al. [18] propose a taxonomy for software evolution, in which they also discuss dimensions relevant to model transformation such as the time and frequency of updates. However, they are much less detailed and at a much higher abstraction level with respect to model synchronization, compared to the dimensions in Section 3.

6 Conclusion and Future Work

In this paper, we characterize the feature model synchronization problem according to dimensions that are applicable to other model synchronization problems and devise a solution to this problem. Important characteristics of our solution are (1) the use of a uniform metamodel for representing feature models, specializations, and configurations; (2) a two-phase approach automatically synchronizing node hierarchy, attributes, and cardinalities, and leaving the violations of cardinalities and additional constraints (such as requires and excludes constraints) to be resolved using the constraint-solving facilities normally available in feature model specialization and configuration. As a bonus, the unification of the metamodel for feature models, specializations, and configurations lead to the development of the check-box view as a uniform interface for editing both specializations and configurations. This is in contrast to our prior work [5], in which the specialization interface was different from the configuration interface.

We have explored two styles of expressing the synchronization logic. A rule-based solution using the *Relations* language of QVT is presented in Appendix A. As experience with this emerging standard and technology is still very scarce, preparing these rules has been an interesting and useful exercise. Unfortunately, we could not test them because no implementation of the language is publicly available as of writing. However, we currently have a Java implementation of our approach to feature model synchronization in an algorithmic style. The implementation is a part of FeaturePlugin [5]. The synchronization facility in FeaturePlugin is used to synchronize feature models with their specializations and configuration that are loaded into the tool. It is also used to synchronize feature models with their metamodels after the metamodels have been modified The use of our synchronization technique is possible in the latter case since metamodels in FeaturePlugin are feature models, too.

In future work, we plan to explore different practical application scenarios for feature model synchronization, including the case where the feature models to be synchronized are distributed. Furthermore, we would like to better understand the kind of changes needing synchronization that are common in practice. On this basis, we would like to explore practical evolution and synchronization strategies. For example, synchronizing feature additions is usually easier than synchronizing removals and relocations. Thus, a conservative evolution strategy could be to avoid changes other than extensions. Alternatively, modifications could be achieved by including both the old and new parts in the new version of a feature model, with the old part marked as obsolete. The feature model could contain constraints and scripts to configure the new parts of a configuration based on the old parts of the configuration. Furthermore, we would like to understand what factors should be looked at to determine when and how often to synchronize. Finally, we plan to explore the application of the constraint based configuration facilities in FeaturePlugin to increase the automation level and support the user in resolving cardinality and other constraint violations during the interactive phase.

Acknowledgements

We would like to thank Ulrich Eisenecker and the anonymous reviewers for their valuable comments on earlier drafts of this paper.

References

1. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, MA (2001)
2. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)
3. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multi-level configuration of feature models. Software Process Improvement and Practice **10** (2005) 143–169 Available from `http://swen.uwaterloo.ca/~kczarnec/spip05b.pdf`.
4. Object Management Group, Inc.: Revised submission for MOF 2.0 Query/View/Transformation RFP (ad/2002-04-10). (2005) QVT-Merge Group, version 2.0, ad/2005-03-02.
5. Antkiewicz, M., Czarnecki, K.: FeaturePlugin: Feature modeling plug-in for Eclipse. In: OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop. (2004) Paper available from `http://www.swen.uwaterloo.ca/~kczarnec/etx04.pdf`. Software available from `gp.uwaterloo.ca/fmp`.
6. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Software Process Improvement and Practice **10** (2005) 7–29
7. Batory, D.: Feature Models, Grammars, and Propositional Formulas. Technical Report TR-05-14, University of Texas at Austin, Texas (2005)
8. Griffin, C.: Model Transformation Framework (2000-2004) Tool available at `http://www.alphaworks.ibm.com/tech/mtf`.
9. Bézivin, J., Jouault, F., Valduriez, P.: First experiments with a ModelWeaver. In: Proceedings of the OOPSLA/GPCE'04 Workshop on Best Practices for Model-Driven Software Development. (2004)
10. Ivkovic, I., Kontogiannis, K.: Tracing evolution changes of software artifacts through model synchronization. In: ICSM. (2004) 252–261
11. Straeten, R.V.D., Mens, T., Simmonds, J., Jonckers, V.: Using description logic to maintain consistency between UML models. In: Proceedings of UML 2003. Volume 2863 of LNCS., Heidelberg, Germany, Springer-Verlag (2003) 326–340

12. Monk, S., Sommerville, I.: Schema evolution in OODBs using class versioning. SIGMOD Rec. **22** (1993) 16–22
13. Ra, Y.G., Rundensteiner, E.A.: A transparent schema-evolution system based on object-oriented view technology. IEEE Transactions on Knowledge and Data Engineering **9** (1997) 600–624
14. Rashid, A.: A database evolution approach for object-oriented databases. In: ICSM. (2001) 561–564
15. Bernstein, P.A., Levy, A.Y., Pottinger, R.A.: A Vision for Management of Complex Models. Technical Report MSR-TR-2000-53, Microsoft Research, Redmond, WA (2000)
16. Melnik, S., Rahm, E., Bernstein, P.A.: Rondo: A programming platform for generic model management. In: Proceedings of ACM SIGMOD, San Diego, California, USA (2003)
17. Sprinkle, J., Karsai, G.: Model migration through visual modeling. In: Proceedings of 3rd OOPSLA Workshop on Domain-Specific Modeling, Anaheim, CA (2003)
18. Mens, T., Buckley, J., Zenger, M., Rashid, A.: Towards a taxonomy of software evolution. In: Proceedings of FWO Network Meeting. Foundations of Software Evolution, Vienna, Austria (2002)

A Synchronization Rules in the QVT Relations Language

The following text shows how the synchronization rules can be specified in the *Relations* language that is described in the latest approved version of the MOF 2.0 QVT proposed standard [4].

The synchronization is expressed as a transformation between a model and its specialization (line 1). When the transformation is called with a specialization as its target, the contained relations (lines 3 and 11) will be enforced (as indicated on lines 6 and 14). The relation `ModelRootFeatureToSpecializationRoot-Feature` requires that if a model root `m` (line 5) and a specialization root `s` (line 6) are connected by a `origin-spec` link (line 8), both roots will also satisfy the `ModelNodeToSpecializationNode` relation (line 9). The latter relation states that for every three matching nodes `m`, `mChild`, and `s`, such that `m.child` = `mChild` and `s` is in `m.spec` (line 13), another node `sChild` must exist (line 14–15). That node `sChild` must be a child of `s`, have `mChild` as its origin, and satisfy `ModelNodeToSpecializationNode` together with `mChild` (lines 17–19). If such a node does not exist, it will be created automatically. The rule also states implicitly that any node in specialization that does not participate in the relationship will be deleted.

```
 1 transformation synchronization(model:FeatureMetamodel, specialization:FeatureMetamodel)
 2 {
 3   relation ModelRootFeatureToSpecializationRootFeature
 4   {
 5     checkonly domain model m:RootFeature{spec=aSpec:RootFeature{}}
 6     enforce domain specialization s:RootFeature{}
 7
 8     when { s=aSpec; }
 9     where{ ModelNodeToSpecializationNode(m, s); } }
10
11   relation ModelNodeToSpecializationNode
12   {
    checkonly domain model m:Node{child=mChild:Node{}, spec=s:Node{}};
    enforce domain specialization sChild:Node{parent=sChildParent:Node{},
                                              origin=sChildOrigin:Node{}};

    where { sChildParent = s;
            sChildOrigin = mChild;
            ModelNodeToSpecializationNode(mChild, sChild); } }
}
```

It is relatively easy to extend the above code to handle the synchronization of node names, attribute types, and attribute values. However, handling node relocation and cardinality changes requires calls to imperative functions, which can be provided as part of the metamodel.

Author Index