Domain Specific Modeling and Analysis

PROEFSCHRIFT

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van Rector Magnificus prof.mr. P.F. van der Heijden,
volgens besluit van het College voor Promoties
te verdedigen op donderdag 13 november 2008
klokke 15.00 uur

door

Joost Ferdinand Jacob

geboren te Haarlem
in 1963

Promotiecommissie

Promotor:	Prof. Dr. F.S. de Boer
Co-promotor:	Dr. M.M. Bonsangue
Referent:	Prof. Dr. H.A. Proper *University of Nijmegen*
Overige leden:	Prof. Dr. J.N. Kok
	Prof. Dr. F. Arbab
	Prof. Dr. W.-P. de Roever *Christian-Albrechts-University of Kiel, Germany*

The work reported in this thesis has been carried out at the Center for Mathematics and Computer Science (CWI) in Amsterdam.

Contents

Chapter 1

Introduction

In the year 2002 I accepted an invitation to work for four years in research projects at the Centrum voor Wiskunde en Informatica (CWI) in Amsterdam. During that time I would investigate possibilities to leverage scientific research projects with the latest computer science knowledge and skills, such as I could deliver. This thesis and the publications herein are the result of that work.

Only my more successful contributions resulted in publications in this thesis, so the reader might get a rosier impression of the application of new techniques to research projects than was actually the case in practice. However, looking back, it was very well worth the effort, and important lessons have been learned that hopefully help improve future projects.

My main projects for CWI where the OMEGA [OME] and Archimate [Arc] projects. A brief description of these projects is in section 1.1.1 for OMEGA and section 1.1.2 for Archimate. In both projects we worked with several types of model data. The model data is usually a static representation of a state or states in the problem domain. Apart from the precise meaning of the static data, a key problem in projects is usually how to specify the transition from one set of data into another, and, if possible, how to do this in such a way that it is formal but also understandable for the various project participants. Preferably such specifications should lead to insights that guide the implementation of tools. In a typical innovative project such as Archimate or OMEGA, as funded by national governments or international bodies, not only scientific researchers are involved but also representatives from industry. The latter do acknowledge the importance of formal proofs and descriptions concerning the problem domain, but the usual scientific presentation of formal results is difficult to comprehend without a thorough background in mathematics and formal methods. This is unfortunate because in this way good results may miss the impact they deserve.

This thesis describes several approaches aimed at bridging the gap between science and industry. A recurring theme is the development of demonstration tools, like web applications, that give an insight into formal methods, and that can serve as an intermediate between pure scientists and others. Due to the nature of such tools[1] and due to the limited time available for their development, it is not always possible to give complete results in this way, but this approach is still important because it makes a full formal result, a project deliverable on paper, more acceptable for the non-scientific partners in a project. Another important theme is communication. It is of paramount importance that the scientists and developers in a project communicate well. A thorough study of the core concepts in a project and an agreement on their names and definition is worth the time invested in it. This is closely related to the design of modeling languages, where a good choice of names and abstractions is essential.

Leveraging domain specific concepts in tools helps to make them more accessible and also helps in shortening the tool development time. In every specialized field there are well-known earlier results that can be re-used without the need for a full proof and corresponding full implementation, like it would be necessary if using a general purpose tool. As a very simplified example consider a tool that helps with automating algebra. This tool would not need to prove everything from the ground up, it can use established axioms like $x + y = y + x$ and it does not have to verify the data types of x and y as long as the tool is not abused. These relaxations makes the implementation much simpler and cheaper to develop. A danger in this approach is that users may *unknowingly* abuse the tool, providing input like `5.0 + "1.2"`, where x is a valid number and y incorrectly is a number in string representation. Such errors are usually easy to spot but they stress the fact that domain specific approaches for tools and modeling, as developed in this thesis, are not primarily intended for delivering full proofs or

[1] Tools like web applications are developed using the latest and popular methods and languages so they are familiar to industry, despite the problem that the latest techniques typically do not have a stable formal basis yet.

fully conclusive results, but they are designed in the first place to help with experimenting and with finding results in a timely and cost-effective manner.

With *domain specific* modeling and analysis, as in the title of this thesis, an approach is intended that leverages as much of the earlier existing work in the problem domain as possible. It does this by re–using parts of the languages and formats, typically XML[XML] vocabularies, that are in common use in the problem domain, in order to be able to model and analyse with a formal basis but in a transparant way and in an affordable way with respect to time and cost contraints, concentrating on the original work. A situation often encountered in projects is that even before an attempt to a formal specification is started, there is already a lot of work done on proof–of–concept systems and tools. Domain specific techniques as developed during my projects capture essential concepts and definitions from this earlier work, and give them a name, an abstraction, that is familiar to the early workers. If a truly formal specification, developed at a later point in time, re–uses these concepts, it is better understood and more readily accepted, even if it does not agree in all aspects with early implementations. To be able to reason about the captured concepts algebraically is desirable, and this is a prime example of the usefulness of the transformation capabilities of the techniques introduced in this thesis. However, the main use of the transformation techniques developed in my projects is to translate from a model with domain specific elements to a model that is suitable for other purposes, like a graphical display for visualization or simulation. This improves the level of understanding and communication considerably.

1.1 Problem statement

In order to introduce the more general problem statements, I will first describe the research projects and the problems encountered there.

1.1.1 The OMEGA project

The OMEGA[2] project was a 3-year IST project, IST-2001-33522 OMEGA, in which the CWI, my employer at the time, participated as a research partner.

[2] (`http://www-omega.imag.fr/index.php`)

The full title of the project was Correct Development of Real-Time Embedded Systems. Besides research partners there were also several industrial partners in the project, and it was sponsored by the European Commission. As a result of this profile the project aimed to achieve not only theoretical results but also some results that have direct practical benefits, as shown by the official aim of the project that is stated *Definition of a development methodology in UML for embedded and real-time systems based on formal techniques* on the project website. The research partners were teams from VERIMAG from France, also acting as project coordinator, Christian-Albrechts-Universität from Germany, University of Nijmegen from The Netherlands, OFFIS from Germany, The Weismann Institute from Israel, and Centrum voor Wiskunde en Informatica from The Netherlands. The industrial partners were EADS SPACE Transportation from France, France Telecom R&D from France, Israeli Aircraft Industries from Israel, and The National Aerospace Laboratory from The Netherlands.

Project OMEGA achieved many results, in the form of publications but also in conferences, workshops and standard contributions to UML 2.0. Because of the pluriformity of the OMEGA work, there were several work packages: Modeling, System Verification, Synthesis, Development Methodology, and Applications. I started working in Modeling, but soon I directed most of my efforts at System Verification. There were also a few contributions for the Development Methodology, such as the coordination language UnCL from chapter six.

In the OMEGA project the Unified Markup Language (UML) is used for modeling, and as a basis for verification. However, UML itself does not have a formal semantics, there is no mathematical definition of UML. This is not an omission in UML but one of its strong points because it gives more freedom in designing and using UML models, which would be harder if UML, for instance, insisted rigidly on a certain model of execution. Instead of incorporating a formal semantics, UML semantics is given by various UML tools, as encountered in the project. There are tools for model building and model checking and simulation.

An important OMEGA result is the development of the OMEGA Kernel Model language. It is a subset of the UML language, capturing core UML concepts that are important for the users in OMEGA. The Kernel Model is used as a reference point for discussions and comparisons of the various verification tools in the project. It incorporates UML extensions for real–time software.

The consistency problem

The OFFIS and VERIMAG teams worked on model–checking tools and the university partners worked with PVS, a theorem prover. These different tools all have different internal formats and unfortunately this resulted in a *consistency problem*, which turned out to be a major challenge. The problem with the tools that we used is that they have internal details that are not part of the model, for instance the use of certain stacks and tables for namespace administration purposes in the software. This leads to practical problems with the consistency of the results acquired with the tools, because the inner workings of the tools differ and it is not feasible to translate semantics from one tool to the other, and to relate these back to the original model, in a consistent way. The Kernel Model semantics is being mangled by adhering to a specific internal tool format and this has damaging effects on the consistency. It is very hard to explain the semantics of a model when using another, specific, semantics.

The CWI team contributed by defining an *abstract* semantics for the Kernel Model. An *abstract* semantics may function as some sort of bridge between different more concrete semantics. In order to remedy some of the consistency problems the CWI team decided to investigate the possibility of a proof–of–concept tool that provides an implementation for the abstract semantics of the OMEGA Kernel Model. The goal was to achieve a complete separation between the event–based operations and the primitive operations. An example of an event–base operation is a method call in OOP software, an example of a primitive operation is the addition of numbers. We also wanted to separate all operations from the scheduling in the executing environment. Chapter 2 presents this work.

1.1.2 The Archimate project

During my stay at CWI I also put a lot of work in the ArchiMate[3] project, a research initiative that aims to provide concepts and techniques to support architects in the visualization and analysis of integrated architectures. The Archimate consortium consists of ABN AMRO, Stichting Pensioenfonds ABP, the Dutch Tax and Customs Administration, Ordina, Telematica Institute, Centrum voor Wiskunde en Informatica, Radboud University of Nijmegen, and the Leiden Institute for Advanced Computer Science. One of

[3] (`http://archimate.telin.nl`)

the results of Archimate is a book publised by Springer with the title *Enterprise Architectures at Work*, and I am one of the authors of that book. The Archimate language developed for enterprise architectures has been adopted as a standard in the Netherlands, Belgium and Luxemburg.

During the project several enterprise architecture description languages were developed, where each language was intended for different stakeholders. A language intended to describe a complete architecture would be too large, and the resulting model would be far too complex. With their own specific language the stakeholders could create a model of an enterprise architecture that would model the parts they were interested in, while abstracting from other parts. Such languages need to capture properties of the system in their bare essence without forcing the architect to include irrelevant detail. The models created in Archimate with these languages were primarily intended for visualisations and simulations, there was no deep investigation into semantics like in the OMEGA project. The work in Archimate was more one of language design than language analysis. An appropriate level of abstraction for the description languages was required, and during the project the languages were subject to change, while searching for such an optimal level in an iterative design process. To complicate matters, since there were different stakeholders in the project with different interests and priorities, their requirements led to very different languages, resulting in a consistency problem similar to that in the OMEGA project. However, in Archimate the consistency problem was of lesser importance, since a unified semantics was not an important goal.

The adaptation problem

The Archimate project developed and used several tools, and the continual rapid changes of the languages posed several problems for the tools that had to work with them. Especially in the early stages there was an *adaptation problem* and this was a bigger problem in Archimate than the consistency problem. The tools had to be able to adapt themselves to new versions of the model languages used, and they had to be able to do that quickly and without too much effort during the course of the project. The model languages used in Archimate were XML[XML] languages, called *vocabularies* in XML terminology, complete with XML *schemas* for the language definitions. If vocabularies are often subject to change, it is best to concentrate on the schemas when developing tools. This is the standard approach when developing XML tools in such circumstances and it was generally followed by the Archimate tools. The goal is to develop tools that take a schema as input, creating a new tool as it where. This virtual new tool can then handle a model in the schema's language, making the original tool rather independent of the specific language used. This approach is not easy and not straightforward, since development has to take place on the basis of a meta–language rather than a final language that can be used immediately for testing purposes. During development it would be hard to envision what the final tool would be like, adding extra uncertainties to the development process.

As one of the participants in the Archimate project, the CWI team drew attention to other XML work being done at CWI and suggested to investigate if new developments there could be used in Archimate. It was at this point that I joined Archimate to see how I could contribute with XML language design and tool development, primarily focusing on the adaptation problem. Part III of this thesis bundles the work in the Archimate project.

The practical problems encountered in the projects lead to the problem statements of my work:

- How can the consistency of project results acquired with various different tools be improved upon? This *consistency problem* is first introduced in section 1.1.1.

- How to develop tools for a project while the underlying modeling languages are still in flux, being designed and changed in an iterative process? This *adaptation problem* is first introduced in section 1.1.2.

- How can project results be communicated well to other project stakeholders, and how can the design of model languages help in this respect?

- How to create a common language of discourse that is still close to the semantics modelled, again with the design of model languages in mind?

- How can modern techniques in software design and programming be leveraged in research projects? Not as a theoretical research topic, but to enhance the project practically, making use of the latest developments. For instance in the area of web-based systems, protocols, and languages, what is hype and what is useful?

- As a specific example of a hyped technique: Can the definition of new XML vocabularies, defining domain specific XML languages, help in a scientific research context?

- What extra contributions to a typical research project, lasting three or more years, are possible with a domain specific approach? Does it open up new ways of getting results, does it bring new insights?

- How valuable are domain specific techniques? Are they only suitable for simulations and demonstrations or can they also help to obtain more formal results?

- Can domain specific tools be developed and used in the timespan of only a few years as is usually the case in a typical research project?

1.2 Objectives

There are several objectives my work tries to achieve. First of all the practical objectives of immediate use in the projects I was involved in: to solve the consistency problem and the adaptation problems from section 1.1, or at least ameliorate them. This is part of the more general objective to find answers to all the other problem statements from that section.

Another, more long-term, objective is to bridge the gap between formal methods and mathematics on the one hand, and software engineering practice on the other hand. This should lead to a better theoretical basis for UML and other models, and ultimately it should lead to software engineering based on sound formal approaches.

Software engineering today roughly uses three types of models. With increasing formality they are: programming language level models (API's) with written comments, standardized diagrams like UML, and formal specifications. This thesis advocates the use of the latter, but its use is still very rare in industry. Reasons why formal specifications are not popular are that many developers would have to be better trained mathematically, and scaling to real–life size systems has not been accomplished often enough. Also, communicating formal models is complicated since there is a problem of choice. There are many formal methods to choose from, and each has its own notations and techniques.

One objective of the use of domain specific techniques in my work is to turn models that have a feeble formal basis into models that are better suitable for formal methods. Even when this does not immediately lead to a full–fledged formal specification, the results do bring much insight and starting points to arrive at such an enhanced specification later.

But perhaps the most important objective of my work has to do with the human aspect: the domain specific techniques help with understanding results, with analysis, with discussions and with communication. Results can be presented with concepts and definitions that have familiar names for everyone involved. The importance of an excellent mutual understanding and a high level of communication is paramount in research projects.

1.3 Approach

To explain the approach of my work, let me start with a summary of it. The research starting point in this thesis is operational semantics, taken as the foundation to understand systems. This is enhanced with term rewriting techniques in order to describe behavior. In order to facilitate the term rewriting, several pattern matching techniques have been developed that are capable of working with modern data formats like XML[XML]. It turns out that these techniques are very useful for dynamic aspects like simulations and visualisations, where they have been succesfully applied, while the underlying operational semantics, or at least the possibility to envision a clear route to such, provides a good understanding of the whole. In what follows I shall give more background to the research approach, using a lot more lines than this summary, but I wanted to present the summary here first to guide the reader with respect to the direction of the work.

A good approach in research based on other research, is to keep the good things and remove the bad, and to add new good things. This seems obvious and this is the approach chosen in my work. However, to use this approach, one has to identify first what is good and what is bad. This may look trivial, but in computer science, which is a relatively young field, good and bad are not so well–established yet and it is hard to get many experts to agree on a certain topic.

I should note here too that all research described here is conducted in the context of projects. This influences the research approach because this means that there have to be things like feasability studies and the research always has to keep project goals in mind.

Part of my research approach was to look for promising new techniques and how they could be applied to the project topics. If a technique looked promising enough then I would design and develop a proof–of–concept tool. A presentation to research partners in the project would then provide valuable input from them about the usefulness and suitability. While not a research question or goal in itself, it was very interesting to find out what others, with a different background, had to say about new development techniques and systems. Not every new technique or approach received a warm welcome, even though it was very popular in the world of development specialists. My domain specific approach was also received with healthy scepticism, but it became readily accepted when application of it in the OMEGA and Archimate projects addressed several research questions and fulfilled several research goals.

In the research projects several different kinds of models were used. Many of these models were UML models like class diagrams, message sequence charts, and use cases. Usually, the complexity of a system is such that many different models are needed to model it. This was also the case in the projects, because the projects wanted to achieve practical results and several real–world systems were under investigation. Each different model is used to describe certain aspects of a system, where only parts of the system important for a certain stakeholder are modeled, and other parts of the system are ignored or modeled in much less detail. There is an analogy with blueprints for a building since there we see different ones for the electricity system, the plumbing and the concrete structure. Such modular design and separation of concerns are all very nice indeed, but it is of paramount importance that the different models are consistent. An ideal plumbing system with very desirable properties is useless if the building is not prepared for it. How to arrive at a consistent set of models is the *consistency problem*.

In order to solve the consistency problem, the UML community devotes much research to meta-modeling techniques. The idea is to define a core model and to be able to derive all other models from it and to be able to integrate existing model types. Unfortunately this does not address a major shortcoming of UML: it being unable to provide consistent analysis tools.

Most existing tools as used in the projects, are based on rather traditional techniques and classic ways of dealing with classes and inheritance and other object oriented paradigms (OOP). The tools themselves are written in traditional and well–known programming languages, like C++ and Java. They are being designed with a rigid top–down design using mostly imperative and OOP techniques. This ties them closely to the model of execution of the programming language chosen and to the intricate details of the compilers used, and these ties are generally incompatible with the chosen core model. Their design and implementation makes the tools rather big and unwieldy to use in novel circumstances, like the introduction of real–time aspects. During my work I kept looking for modern techniques that could be of assistance here. I was also looking for small tools rather than big ones, looking for a combination of small tools that could be better than their sum. Another aspect is the way that tools may exchange models. In order to be able to exchange models an XML vocabulary has been designed by a consortium of UML users, and this XML vocabulary was called XMI. XMI can be seen as a common collection of structures and names and definitions that the various UML tool vendors agreed upon. This leads to the idea of also using it as a basis for analysis techniques and even for formalisations of behavior, since a recurring problem in these is often the establishment of a common language of discourse and good set of definitions that is commonly understood. XMI is very complete but because of this unfortunately also very complex, and less complex solutions were needed for analysis and for dealing with behavior. In the projects I kept looking around for new developments to find such solutions.

With respect to useful specific “latest” computer science techniques, I have used a *dynamic programming language* to be able to provide an executing environment for the various models, and I have chosen *data–centric techniques* to arrive at open and transparant systems with interchangeable data. The choice of a popular modern dynamic programming language proved to work out well, since it was capable of providing more flexible solutions in a shorter time span than would have been possible with traditional languages like C++ and Java. It also provided us with very up–to–date libraries for working with XML and other structured data, where we would have had to wait a significant time period, like months, for similar C++ and Java libraries.

I would like to note here, perhaps again, the importance of taking the existing original structured data, such as XMI, as starting point. This increases the level of trust and understanding in the new, smaller, more formal, model. It also makes validation easier and it can better be verified how the new model relates to the old situation. It is important that familiar names re-appear, familiar structures re-appear, and in the case of an executing model the same familiar execution steps can be recognized. In an ideal situation one should start with formal specifications, and be free in the choice of names and concepts, but in reality this is not always possible. For instance, Project Management may have decided to use certain UML tools, or to use certain existing software libraries, for reasons that are not always disclosed and anyway beyond the scope of this discussion. Such circumstances however have to be accepted as part of project reality, and I have encountered them in every single project I have been involved in during my twenty–five years in ICT.

While the UML community spends much effort on meta–modeling techniques, my approach concentrates on the integration of models by trying to find similarities while avoiding as much as possible having to put a tree hierarchy on the models. Complementary to the meta–modeling, which is a top–down approach, the domain specific techniques give a bottom–up approach to arrive at an adequate model core. Or, if a single core can not be achieved, the approach still provides methods to relate models to each other, based on an improved mutual understanding of domain specific notions and concepts.

1.4 Working with XML and other structured data

Models, formulas and other data are nowadays often expressed in XML [XML]. It is believed that next–generation programming systems will have computer programs stored as XML or XML–like documents, to increase interoperability, the goal being that data and meta–data can be represented and processed uniformly [Wil05]. XML is seeing an continually increasing use as the format of choice for modeling language, and it is now the most popular choice. A large part of the thesis is about using XML, and about an XML extension called the Rule Markup Language (RML), described in Chapter 3, in particular. It is shown how to define XML languages, with the emphasis on XML for formal methods, and approaches and methodologies are discussed. With RML it is possible to define rule–based transformations of XML in XML itself, and more importantly, this can be defined in the XML *vocabulary* for the topic at hand itself. RML uses the general technique of *pattern–matching* and *variable–binding*, known from the world of regular expression tools like Perl, where in the case of RML the patterns match XML–parts. These patterns are also expressed with reuse of the domain specific XML vocabulary of choice. Variable bindings with domain specific data can be stored and used at a later time to modify or create other data. An important result in my work is that the freedom given by this approach makes it possible to study and demonstrate formal methods and their applications to models expressed in XML without any restrictions due to the design or implementation of the underlying tools such as modelcheckers and theorem provers.

Besides RML I introduced two other XML techniques to the projects I was involved in: AML [Jaca], see section 8.3, a simpler representation for XML for presentation purposes that is also used to be able to create XML with a simple text-editor, and OOXML, an object-oriented databinding for XML in a high–level scripting language. Like RML, AML and OOXML proved to be very useful to get various work with XML done in a timely fashion in typical research projects.

The pattern–matching and variable–binding approach taken for the XML case with RML can also be applied to other structured data, like text–based notations for formulas. For this purpose ATL has been developed, a wildcard-matching technique for structured text with an as-simple-as-possible design that has a much lower learning curve than typical classical regular expression libraries like those found in Perl, making it applicable without having to learn a full programming language. As a practical example of ATL, a web application is developed that assists with proofs using the tableau method, and a non-trivial proof is derived for the OMEGA project.

1.5 Structure of the thesis

Because of the nature of my work in the projects, the following chapters in this thesis are a number of publications, where every paper forms a chapter by itself. So far the presentation in this thesis has been from abstract to more concrete, but in this section I will revert to a more general bird's eye view of the publications, relating them to each other and to the problem statements, the objectives, and the chosen approach.

There are several scientifically refereed publications, they are:

- Chapter 2. RML and its application to UML. Author: Joost Jacob. Published by Springer in the ISOLA conference proceedings in the series Lecture Notes in Computer Science, volume 4313, year 2006. [Jac04a]

- Chapter 4. The OMEGA Component Model. Author: Joost Jacob. Published by Springer in the journal Electronic Notes in Theoretical Computer Science, volume 101, year 2004, pages 25-49. [Jac04b]

- Chapter 8. Enterprise Architecture Analysis with XML. Authors: Frank de Boer, Marcello Bonsangue, Joost Jacob, Andries Stam, Leendert van der Torre. Publised by the IEEE Computer Society in the 2005 HICSS conference proceedings. [dBBJ$^+$05]

- Chapter 9. A Logical Viewpoint on Architectures. Authors: Frank de Boer, Marcello Bonsangue, Joost Jacob, Andries Stam, Leendert van der Torre. Publised by the IEEE Computer Society in the 2004 EDOC conference proceedings. [dBBJ$^+$04]

- Chapter 10. Using XML Transformation for Enterprise Architecture. Authors: Frank de Boer, Marcello Bonsangue, Joost Jacob, Andries Stam, Leendert van der Torre. Publised by Springer in the ISOLA conference proceedings in the series Lecture Notes in Computer Science, volume 4313, year 2006. [SJdB$^+$04]

The following chapters are grouped into three parts. Part I introduces RML and its tool support, and contains a paper with results in the OMEGA project. Part II is about work on component models in OMEGA and in distributed environments and introduces another pattern matching technique similar to RML as it was used in OMEGA. Part III is also about models and analysis, but here it is enterprise architectures that are modeled in the Archimate project and RML returns as it is used for their analysis.

Part I is named **RML, a tool for model analysis**. In chapter 2 it starts with a paper titled *A Rule Markup Language and Its Application to UML*[Jac04a]. In this paper RML is introduced and an application to UML models is exhibited. This was my first example where a domain specific technique was succesfully applied. Chapters 2 and 3 contain the main introduction to RML. In the OMEGA work described in chapter 2 we were able to demonstrate that models could indeed be executed based on the abstract semantics we designed. This was important since the abstract semantics of the OMEGA Kernel Model helped to relate the other results in the project to each other. Also in Part I, in chapter 3, is the RML Tutorial, with examples of all kinds of XML transformations and how to perform these with RML. Part I lays a foundation for the rest of the thesis. RML was used in the majority of my work, and the pattern matching and term–rewriting ideas from RML did play an important role in the rest of it.

Part II has the title **Component Models and Analysis** and consists of four chapters, chapters 4 to 7. Chapter 4 is a paper that reflects the CWI contribution to the OMEGA project with respect to component modeling in UML. Several ideas from the paper can be found in UML standards that appeared later, starting with UML 2.0, for instance the way to model component ports. In Chapter 5 is an OMEGA publication called *Component Coordination in UML*. It has soms overlap with Chapter 4 because it also uses the OMEGA modeling, but it is focusing on *coordination* of components. Chapter 6 is a publication from the Software Engineering department of CWI, SEN report E0511 from 2005, titled *The unified coordination language UnCL*. It is a fusion of my work in OMEGA on components and the work of my colleague Juan Guillen Scholten at CWI on distributed channels, resulting in a coordination language. Chapter 7 is a CWI publication titled *ATL Applied to the Tableau Method*. This paper shows a novel technique that was used in OMEGA to aid in the proof of a software property. The software was modeled with the OMEGA kernel model from chapter 4 but instead of transforming model data in XML, here we wanted to transform formulas with statements about the models. The ATL approach resulted in additional insights, enhancing earlier proofs that were performed in OMEGA using more conventional methods. Part II shows a progression from static models to more dynamic models and their analysis, with a few digressions in order to explain the techniques used.

Part III consists of three papers on enterprise architectures and is titled **Modeling and Analysing Architectures**. Chapters 8 and 9 are papers with the titles *Enterprise Architecture Analysis with XML* [dBBJ$^+$05] and *A Logical Viewpoint on Architectures* [dBBJ$^+$04]. Chapter 10 is the paper titled *Using XML Transformation for Enterprise Architecture* [SJdB$^+$04]. With respect to my contribution to these papers, the results build on the experience gained with models and analysis in part I and part II, but since they are all papers from the Archimate project and their common theme is enterprise architectures, these papers are presented last and bundled together.

My contribution to Archimate consists of XML language design for business processes and their visualizations and simulations and especially the

RML tool for performing transformations on models in XML. Tools for visualization or simulation use RML for the necessary XML transformations. With RML it becomes practically feasible to tune XML languages to the desired goals in an iterative process, while still using the same tool for visualization and the like, without having to recompile or rebuild the tool. The data–centric nature of the RML tools is helpful in this respect: as much logic and behavior as possible is stated in rules and scripts, removing the need to program them in a much more lower level programming language. Language changes are easy to incorporate in the RML rules, since those rules are as close to the language itself as we could design. RML makes it readily possible to transform systems described in one language to another, to analyse and query systems, and RML also provides an executable framework wherein the dynamic behavior of systems defined in the languages can be quickly tested and analysed, before committing too much resources to the development of fully optimized and specialized tools. The RML contribution to the Archimate project is also described by me in chapter 10 of the Springer book *Enterprise Architecture at Work* [ea05].

Since several chapters contain complete papers as published, some chapter contents have a little overlap. This overlap is not removed but preserved in order to support the reader when reading a chapter by itself, without having to direct the reader to other parts of the thesis, for instance for a short introduction of RML.

1.6 Conclusion

The most successful domain specific approaches in the research projects I have contributed to were the development of new XML vocabularies for modeling and analysis purposes, and the RML and AML tools for handling the new XML that was created with the new vocabularies.

Developing new XML vocabularies has been beneficial in both the OMEGA and Archimate projects. The new XML vocabularies formed a basis for tool development and also for discussions of various data–related topics, both static and dynamically. AML made it possible to use the new XML vocabulary in such discussions in a readable form. For instance, discussions about the flow of events in the OMEGA kernel model could be illuminated with simple classes and objects represented in AML and they could even be dynamically executed with a tool for demonstration purposes. The domain specific techniques did help to achieve a much higher level of communication and understanding between the various project members.

Looking back, especially the development of RML was very helpful in producing results in the OMEGA and Archimate projects. The existing XML tools that were in use in industry at the time were too cumbersome and producing tools with them would take too long in a research project setting. However, today RML has not attained a top–rate status when it comes to XML tools. Reasons are that the CWI research institution where it was developed is not a commercial software house, meaning it has no incentive nor facilities to produce industry–strength competitive software, and it does not have a marketing department that can draw attention to its products. There is also the fact that the main RML virtues are its simplicity and minimalism, and those virtues do not have much marketing value in today's ICT world. Anyway, it is not the tool itself, but its underlying principle of using a domain specific approach, that I consider an important result of my work.

Domain specific languages and models and methods deserve attention from the scientific world. They are popular and they are found everywhere. As an example, consider the HL7 [SRMM00] [7] language that is used in the healthcare domain. The aim is to support hospital workflows through electronic messages exchange between administrative, logistical, financial as well as clinical processes (for instance to send patient data to a radiology department). While it initially used a proprietary (non-XML) syntax, the most recent version uses only XML as a syntax for messages.

Almost all hospitals in The Netherlands use HL7 messages and documents for exchanging medical information. A large number of tools are available for developers, implementers and users of HL7, mostly concentrating on simulation, editing, viewing and validating the XML specific vocabulary of HL7. For these tools, either their formal basis seems feeble, or, as in the case of commercial products, their formal basis is undisclosed. Most of the tools that are available commercially to work with HL7 are complex, often not satisfactory, cumbersome and require users to follow courses to even learn to work with them.

In my CWI research projects I concluded that several small tools may together produce a better result than one large system, on the condition that their results are consistent. But this requires more time spent on design and discussions, and a less strict product–manager–like attitude. In my opinion, theoretical and hard–core computer scientist are definitely able to give a valuable contribution to the use of various domain specific languages. But they are sometimes not invited when I feel they should be. As a result, several real–world domain specific languages have a basis that is not as formal as would be desirable. And the other way around, computer scientists are sometimes not interested in a domain specific language project, being afraid of being dragged into tool development with little scientific value. This situation is unfortunate for both sides, and I feel there are many improvements possible, for instance the use of domain specific techniques with a design like I used for RML.

Why do projects spend so much time and effort to define domain specific languages instead of first defining a formal specification and then building a language on top of that? With a formal specification in hand, designing a domain specific language is much more robust and also simpler, even when the foqmal specification is only halfway ready. There are several reasons. Unfamiliarity of managers, directors, and other decision makers with formal methods is one. Scarcity of mathematically schooled developers is another. Yet another reason is that there is often an earlier body of work, for instance an existing implementation of part of the desired functionality, and management decides that it is cost–efficient and wise to reuse it. All such reasons obstruct a good design of a domain specific language. This is unfortunate, since it is my experience that especially in the early stages of a project, results are obtained faster when working with a well–designed domain specific language for the data rather than by taking the traditional route of defining the data in a full fledged programming language, for instance an object oriented class library in Java or a complex datastructure in C. And still, this is what happens often when the decision is made to reuse existing software or an existing tool, thereby making a formal basis problematical.

Modifications to a data design are easier when it is more separated from the tool implementation, and such modifications are frequently needed in the early stages of a project. A modification like changing a naming convention in the data may seem unsignificant but it is not, because the data language serves as a language of discourse in project discussions. A new naming convention for a group of data elements is much simpler to implement within a domain specific language than in an object oriented class library, and this is just another example of why it is advantageous to use a domain specific approach.

Most research questions from chapter one, the Introduction, have been anwered in the publications in the later chapters. It has been found possible to introduce new techniques in scientific research projects in a beneficial way. Mainly to produce tools for visualisations and simulations, but also contributing in a more fundamental way and resulting in proof–of–concept tools. On several occasions the tool developement led to fruitful discussions and new ideas. Usage of XML and the design of new XML vocabularies proved to be valuable. The development of new small tools to work with the XML also proved to be worthwhile, working on the XML itself or for instance to translate from XML to PVS. It was sometimes possible to combine a set of small tools resulting in a whole that was better than their sum. This is reminding us of the well–known ways a combination of tools in the UNIX world would be used to produce new tools, an art that has become less popular in these days of big computer languages like Java and C# and their massive development environments. Making use of new dynamic programming languages and data–centric techniques, we were able to develop such new small tools within the timespan of the projects, and here I feel that it was important that there were not too much restrictions on the implementation. It was important that the programmer was free in the choice of a programming language and in the design of the tools. Programmers need freedom to be creative and productive, and it seems that the better the programmer, the more freedom is necessary. On first sight, this principle advises against the use of formal specifications, but I believe this is not the case. If the formal specification is able to stay close to the world of the programmer, using concepts and definitions the programmer is familiar with, then the insights acquired from the mathematics are a joy to work with. The development of domain specific techniques and their application helps to bring formal methods closer to the many existing and popular domain specific languages that are already being used on a large scale but lack a real formal basis.

Finding new techniques, determining their usefulness, and introducing them to projects, remains a considerable task. Some new techniques proved to be helpful, like the XML modeling that could quickly yield new tools, while other new techniques turned out to be mostly hype and they could not withstand scrutiny by scientific minds.

Part I

RML, a tool for model analysis

Chapter 2

RML and its application to UML

Author: Joost Jacob

2.1 Introduction

The work in this paper was initiated and motivated by work in the IST project OMEGA (IST-2001-33522, [OME]) sponsored by the European Commission. The main goal of OMEGA is the correct development of real-time embedded systems in the Unified Modeling Language [SWB03]. This goal involves the integration of formal methods based on model-checking techniques [BDJ+03] and deductive verification using PVS [ORR+96].

The eXtensible Markup Language XML (XML [XML]) is used to encode the static structure of UML models in OMEGA. The XML encoding is generated by Computer Aided Software Engineering (CASE) tools; it captures classes, interfaces, associations, state machines, and other software engineering concepts. The OMEGA tools for model-checking and deductive verification are based on a particular implementation of the semantics of the UML models in a tool-specific format ([BGM01], [DJPV03], [ORR+96]). This complicates interoperability of such tools. In order to ensure that these different implementations are consistent, a formal semantics of UML models is developed in OMEGA in the mathematical formalism of transition systems

[Plo81]. However, it still requires considerable effort to ensure that these different implementations are indeed compatible with the abstract mathematical semantics. Some of the motivation for RML came in helping with this effort. Since the models produced by the CASE tools are encoded in XML it was a natural choice to look for an XML transformation technique instead of encoding a model and semantics in a special-purpose format. Simulating and analyzing *in* XML adds the interoperability benefit of XML and the many available XML tools can be used on the results.

In this paper a general-purpose method for XML transformations is introduced and its application to the specification and execution of UML models. The underlying idea of this method is to specify XML transformations by means of rules which are formulated in a problem domain XML vocabulary of choice: the rules consist of a mix of XML from the problem domain and the Rule Markup Language (RML, Sect. 2.3). The input and output of a transformation are pure problem domain XML; RML is only used to help to define transformation rules. The RML approach re-uses the problem domain XML as much as possible, with a "programming by example" technique. With this rule–based approach it becomes possible to define transformations that are very hard to do when using for example XSLT [XSL], the official W3C [W3C] Recommendation for XML transformations, as discussed in Section 2.2.1.

The RML tools are available as platform-independent command–line tools so they can easily be used together with other tools that have XML as input and output.

RML is not trying to solve *harder* or *bigger* transformations than other approaches. Instead of concentrating on speed or power, RML is designed to be something that is very *usable* and interoperable. Experience in several projects has shown that programmers can learn to use RML in only a few hours with the tutorial in Chapter 3, and even non-programmers put RML to good use. With respect to the RML application to UML models, only knowledge of XML and RML suffices to be able to define and execute their semantics.

As such, RML provides a promising basis for the further development of XML-based debugging and analysis tools for UML models.

XML itself is not intended for human consumption, but we have developed the ASCII Markup Language (AML) representation that helps considerably in this respect. The example model in this paper is presented in AML because AML is more readable than XML, but otherwise equivalent for this purpose.

More details about AML and an AML to XML translation, and back, are available at [Jaca].

Plan of the paper The next section starts with describing XML. Section 2.3 presents RML as a new approach to solve XML transformation problems and describes how to use RML for defining transformation rules. Section 2.4 shows examples of applications of RML, the main example being an application that results in executable UML models. The conclusion and a discussion of related work is in Sect. 2.5.

2.2 XML and XML transformations

With XML, data can be annotated and structured hierarchically. There are several ways to do this and there is no single best way under all circumstances: designing good XML vocabularies is still an art. For instance, suppose you want to describe a family in XML: a grandmother named Beth, a father named John, a mother name Lucy and son named Bill. One way to do this is:

```
<family>
    <grandma name="Beth" />
    <father name="John" />
    <mother name="Lucy" />
    <son name="Bill" />
</family>
```

The example shows five different XML elements: `family`, `grandma`, `father`, `mother`, and `son`. The XML hierarchy is a tree, with nodes called XML elements, and there has to be one and only one XML element that is the root of the tree, in the example the `family` element. An XML element consists of its name, optional attributes and an ordered list of subelements, where a subelement can also be a string. Attributes of XML elements are mappings from keys to values, where the keys are text strings and the values are text strings too.

A string enclosed with angle brackets is called a tag. A minimum tag only contains the element name, like the `<family>` in the example. The element name is not the only thing that can appear between the angle brackets, there can also be attributes like `name="John"` in the example. Attributes consist of the attribute name, an `=` and the attribute value (a text string) enclosed in double quotes.

An XML element that does *not* contain other elements, a so called empty element, has its tag closed by an /, as in <X />, where X is the element name. An XML element that has children consists of two tags: one for the element name (and its attributes), and one for closing the element after its children. In the example the family element is the only element with children. There are several rules that define if XML is *well formed*, for instance every opening tag <X> has to be closed by a closing tag </X>, and these rules can be checked by tools.

But the XML in the example does not reflect the tree like structure of the family. Another way is:

```
<family>
    <female>
        <name>Beth</name>
        <male marriedTo="Lucy">
            <name>John</name>
            <male>
                <name>Bill</name>
            </male>
        </male>
    </female>
    <female marriedTo="John">
        <name>Lucy</name>
    </female>
</family>
```

Here Beth is not the value of an attribute but it is the text content of a name element. The structure of this example may better indicate that Beth is the mother of John, but the XML is more verbose than the first example.

An XML vocabulary can be formally defined in a DTD (see the XML Specification in [XML]) or an XML Schema [XMS], both W3C Recommendations. There is also an ISO standard for defining vocabularies called RelaxNG [Cla01]. The definition can express that for instance every female in the example must have a name child and can have optional female or male childs. With such a definition, called schema, there are XML tools available that can *validate* if XML is conforming to a schema. Note that validating is different from checking well-formedness. It is possible to refer to the definition of the vocabulary used from inside XML, and there are many more XML concepts that can not be discussed here due to lack of space, for which I refer to the XML Specification [XML].

A schema only defines syntax, the *meaning* of the XML constructs defined has to be defined somewhere else. The W3C is working on standard ways for doing this (viz. the Semantic Web project) but currently this is usually done in plain text documents. The family from the example has a tree like structure so in this respect it is an easy example to describe in XML that also has a tree structure. But a tree is not the only kind of structure that can be modeled conceptually with a schema. Other structures can also be modeled in XML because the XML elements can refer to each other by means of cross-references with identifiers, as in the second example with the `marriedTo` attributes.

A lot of XML vocabularies have been designed in recent years, for all kinds of problem domains. Often these vocabularies are W3C Recommendations, like RDF, XSLT and MathML [Mat03]; another example is XMI [XMI] as developed for UML [SWB03] by the Object Management Group [OMG]. The OMEGA project works with XMI and other XML vocabularies for software.

In general, when having a structure stated in XML data, a dynamics of the structure can be captured by rules for transforming the XML data. The rules that define XML transformations can be stated in XML itself too, but the problem domain XML vocabulary will usually not be rich enough to be able to state a rule. For this it has to be combined with XML that is suitable for expressing transformation rules, containing for example constructs to point out what to replace with what, and where. Section 2.2.1 shows how an XML vocabulary that is different from the problem domain vocabulary can be used, which is the current way of doing transformations in industry, and Sect. 2.3 shows the new RML approach that is based on extension.

2.2.1 XSLT

Extensible Stylesheet Language Transformations (XSLT, [XSL]) is a W3C recommendation for XML transformations. It is designed primarily for the kinds of transformations that have to do with visual presentation of XML data, hence the *style* element in the name. A popular use of XSLT is to transform a dull XHTML page to a colorful and stylized one. Or to generate visualizations from XML data. However, nowadays XSLT is being used more and more for general purpose XML transformations, from XML to XML, but also from XML to text. In the OMEGA project we have used XSLT to do transformations of software models (UML models stated in XMI [XMI]) resulting in models encoded in other XML vocabularies and also resulting in textual syntax like PVS [ORR$^+$96]. The static structure of the models was transformed. But when we wanted to capture the semantics of execution of these software models we found that XSLT is not very usable for the particular kind of transformations that describe dynamics. These transformations use a match pattern that is distributed over several parts of an XML tree, whereas the matching technique used in XSLT is designed to match in a linear way, from root to target node in a tree. This linear matching is not suitable for matching of a pattern with several branches.

For instance, matching duplicate children of an element is very hard with XSLT. The MathML expression

```
<math>
    <apply>
        <and />
        <ci>p</ci>
        <ci>p</ci>
        <ci>q</ci>
    </apply>
</math>
```

meaning $p \wedge p \wedge q$ in propositional logic, is logically equivalent to

```
<math>
    <apply>
        <and />
        <ci>p</ci>
        <ci>q</ci>
    </apply>
</math>
```

meaning $p \wedge q$. In the MathML the `<ci>` element is used for pointing out constant identifiers and the `apply` element is used for building up mathematical expressions. Suppose we would like to transform all $p \wedge p \wedge q$ into $p \wedge q$, where p and q can be anything but two p's in an expression are equal. To perform such a transformation a tool has to look for a pattern with two identical children and then remove one of the children. Since XSLT is a Turing–complete functional programming language, it *is* possible to do this transformation, but XSLT templates for these kinds of transformations are extremely long and complex. XSLT simply was not designed for these kinds of transformations; the designers did not feel much need for them in the webwide world of HTML and webpublishing. The MathML+RML rule for removing duplicate children is simple, it is one of the examples in the RML tutorial in Chapter 3.

2.3 RML

This section first introduces the idea of *XML wildcard elements.* After that the RML syntax is introduced in Sect. 2.3.2, before Section 2.3.3 describes the RML tools.

2.3.1 XML Wildcard Elements

The transformation problem as shown in Sect. 2.2.1 reminds one of the use of wildcards for solving similar problems in text string matching. The idea of using an XML version of wildcards is a core idea of our method and of this paper. The idea is to define XML notation for an XML version of constructs like the `*` and `?` and `+` and others in text–based wildcards as in Perl regular expressions, and then using these constructs for matching and variable binding. These constructs are called *XML wildcard elements.* They consist of complete XML elements and attributes as can be formally defined with a schema, but they also consist of *extensions* for denoting wildcard variables *inside* a problem domain XML. These variables are just like the variables as they are used in the various languages available for text–based wildcard matching, for instance Perl regular expressions. The variables have a name, and they are given a value when a match succeeds. This value can then be used in the output of a transformation rule.

2.3.2 The RML syntax

RML rules are stated in XML. The basis of a rule is that in the antecedent of the rule the input is matched, and then whatever matched is replaced by the consequent of the rule.

RML was designed to be *mixed* with any problem domain XML, to be able to define transformations while re-using the problem domain XML as much as possible. RML is a mixture of XML elements, conventions for XML-attribute names, and conventions for attribute values, to mix in with XML from the problem domain vocabulary at hand. RML introduces some new XML elements and uses an element from XHTML. From XHTML only the `div` element is used and it is used to distinguish a rule, the antecedent of a rule, and the consequence of a rule by means of the `class` attribute of the `div` tag. We use the `div` tag from XHTML for reasons that have to do with a presentation in browsers.

The Table in Fig. 2.1 lists all the current RML constructs with a short explanation of their usage in the last column. These have been found to be sufficient for all transformations encountered so far in practice in the projects where RML is being used. An `X` in the XML tags can be replaced by a string of choice. The *position* that is sometimes mentioned in the explanations is the position in the sequential list that results from a root-left-right tree traversal of the XML tree for the rule. It corresponds with how people in the western world reading an XML document encounter elements: top-down and left to right. A position in the rule tree corresponds with zero or more positions in the input tree, just like the `*` in the wildcard expression `a*b` corresponds with `c` on input `acb` and with `cd` on input `acdb` and with nothing on input `ab`. With the constructs in the Table in Fig. 2.1 the user can define variables for element names, attribute names, attribute values, whole elements (with their children) and lists of elements. An XML+RML version of the `a*b` wildcard pattern is

```
<a /> <rml-list name="Star" /> <b />
```

and this can be used as part of the antecedent of an RML rule that uses the contents of the `Star` variable in the consequent of the rule. The `a` and `b` elements are from some XML vocabulary, the `rml-list` element is from RML and described in the Table in Fig. 2.1. Section 2.4.1 shows a small but complete RML rule. Examples of input and rules take up much space and although we would have preferred to present more rule examples here now, there is simply not enough space to do that and we strongly invite the reader to look at the examples in the RML tutorial in Chapter 3 where there are examples for element name renaming, element replacing, removing duplicates, copying, attribute copying, adding hierarchy and many more.

It is easy to think of more useful elements for RML than in the Table, but not everything imaginable is implemented because a design goal of RML is to keep it as simple and elegant as possible. Only constructs that have proven themselves useful in practice are added in the current version.

The execution of a rule consists of binding variables in the matching process, and then using these variables to produce the output. Variable binding in RML happens in the order of a root-left-right traversal of the input XML tree. If an input XML tree contains more than one match for a variable then only the first match is used for a transformation. The part of the input that matches the rule antecedent is *replaced* by the consequent of the rule. If a rule does not match then the unchanged input is returned as output. If a rule matches input in more than one place and you want to transform all matches then you will have to repeat applying the rule on the input until the output is stable. There is a special RML tool called `dorules` for this purpose.

2.3.3 The RML tools and libraries

Open-source tools and libraries can be downloaded and also the RML tutorial in Chapter 3 is available online. The tutorial contains information about installing and running the tool, and there is also more technical information on the website, for instance about a matching algorithm. The tools and libraries have been successfully used under Windows, Linux, Solaris, and Apple.

A typical usage pattern of the `applyrule` command–line tool is `$ python applyrule.py --rule myrule.xml --input myinput.xml` that will print the result to console, or it can be redirected to a file.

The rule and the input are parameterized, not only as command–line parameters but also in the internal `applyrule` function; this makes the tool program also usable as a library and thus suitable for example for programming a simulation engine. There is an interface to additional hook functions so tools can be extended, for instance for programming new kinds of constraints on the matching. However, the set of RML constructs in the current version has proven to be sufficient for various XML transformation work, so adding functions via the hooks will normally not be necessary. An example of when it *is* desirable to add functions is when a tool designer for example wants to add functionality that does calculations on floating point values in the XML. The RML tools are written in Python [vR95] and the Python runtime can use the fast `rxp` parser written in, and compiled from, C, so the XML parsing, often the performance bottleneck in such tools, is as fast and efficient as anything in the industry. If the rxp module is not installed with your Python version then RML automatically uses another XML parser on your system.

The RML tutorial in Chapter 3 also describes a very simple XML vocabulary for defining RML recipes, called Recipe RML (RRML), and a tool called `dorecipe` for executing recipe–based transformations. RRML is used to define sequences of transformations and has proven itself useful in alleviating the need for writing shell scripts, also called batch files, containing sequences of calls to the RML tools.

2.4 RML examples

The main example presented in this paper is the executable specification of the semantics of UML models in XML and this is the topic of Section 2.4.1. Section 2.4.2 briefly mentions other projects wherein RML is applied.

2.4.1 Executable UML models

The application of RML to the semantics of UML models and its resulting execution platform is based on the separation of concerns betweeen coordination/communication and computation. This exploits the distinction in UML between so-called triggered and primitive operations. The behavior of classes is specified in UML statemachines with states and transitions, and every transition can have a trigger, guard, and action. A transition does not need to have all three, it may for example have only an action or no trigger or no guard. Triggered operations are associated with events: if an object receives an event that is a trigger for a transition, and the object is in the right location for the transition, and the guard for that transition evaluates to True, then the action that is specified in the transition is executed. The triggered operations can be synchronous (the caller blocks until an answer is returned) or asynchronous. Events can be stored in event queues, and the queues can be implemented in several ways (FIFO, LIFO, random choice, ...). There are also primitive operations: they correspond to statements in a programming language, without event association or interaction with an event mechanism. The primitive operations are concerned with computations, i.e. data-transformations, the triggered operations instead are primarily used for coordination and communication. More details can be found in [DJPV03].

This distinction between triggered and primitive operations and the corresponding separation of concerns between coordination/communicaion and computation is reflected in the RML specification and execution of UML models which delegates (or defers) the specification of the semantics and the execution of primitive operations to the underlying programming language of choice. This delegation is not trivial, because the result of primitive operations has to be reflected in the values of the object attributes in the XML, but the details of the delegation mechanism can not be given here due to a lack of space.

In our example the problem domain is UML and we will use a new XML vocabulary that is designed for readability and elegance. This language is called *km*, for kernel model; a RelaxNG [Cla01] schema is at http://-homepages.cwi.nl/˜jacob/km/km.rnc.

The online example is a prime sieve, it was chosen because it shows all the different kinds of transitions and it has dynamic object creation. It generates objects of class `Sieve` with an attribute `p` that will contain a prime number. But the user can edit the example online or replace it with his or her own example, if the implementation language for actions and guards is the Python programming language. A similar application can be written for the Java language, and UML models from CASE tools can be translated automatically to the km language. The example can be executed online in an interactive webapplication on the internet at [Jacd]. In the km application the user fills in a form with an object identity and a transition identity, and pressing a button sends the form to the webapplication that performs the corresponding transition. Instead of a user filling in a form, a program can be written that calls the website and fills in the form, thus automating the tool. We did so, but for this paper we consider a discussion of the automated version out of scope.

The km language defines XML for class diagrams and object diagrams. The classes consist of attribute names and a statemachine definition. The statemachines have states and transitions, where the transitions have a guard, trigger and action like usual in UML. The objects in the object diagram have attributes with values and an event queue that will store events sent to the object. An example of an object is

```
objectdiagram
    obj class=Sieve id=2 location=start target=None
        attr name=p value=None
        attr name=z value=None
        attr name=itsSieve value=None
        queue
            op name=e
                param value=2
```

where the object is of type `Sieve`, finds itself in the `start` state of the statemachine of the `Sieve` class, and has an eventqueue with one event in it with name `e` and event parameter `2`.

A detailed description of the km language and its design would take too much space here, but the interested reader who knows UML will have no trouble recognizing the UML constructs in the models since the km language was designed for readability.

In the km language the event semantics is modelled, but the so-called *primitive* operations that change attribute values are deferred to a programming language. So the models will have event queues associated with objects and executing a model will for example show events being added to queues, but operations that are not involved with events but only perform calculations are stored in the model as strings from the programming language of choice. Such an operation can be seen in the example as

```
transition id=t3
    source state=state_3
    target state=state_1
    action
        implementation
            """x = x + 1"""
```

where we see a transition in the statemachine with an action, the statement executed by the programming language (Python in this case) is `x = x + 1`. Transitions can also have a guard with an expression in a programming language, also encoded as text content of an `implementation` element.

We can now show a simple example RML rule.

```
<div class="rule" name="set location">
  <div class="antecedent">
    <obj id="rml-IDOBJ" location="rml-L" target="rml-T" rml-others="rml-O" >
      <rml-list name="ObjChildren"/>
    </obj>
  </div>
  <div class="consequence">
    <obj id="rml-IDOBJ" location="rml-T" target="None" rml-others="rml-O">
      <rml-use name="ObjChildren"/>
    </obj>
  </div>
</div>
```

This is a rule that is used after a transition has been taken successfully by an object modeled with km. With this rule the `location` attribute of the object is assigned the value of the `target` attribute and the `target` attribute is set to `None`. An example of the effect of the rule would be that

```
<obj id="id538" location="state_3" target="state_5" ... >
  <queue>
    ...
  </queue>
</obj>
```

is changed into

```
<obj id="id538" location="state_5" target="None" ... >
  <queue>
    ...
  </queue>
</obj>
```

for an object with identifier `id538`.

When applying this rule, the RML transformation tool first searches for an `obj` element in the input, corresponding with the `obj` element in the `antecedent` of the rule. These `obj` elements match if the `obj` in the input has an `id` attribute with the value bound to the RML `IDOBJ` variable mentioned in the antecedent, in the example this value is `id538` and it is bound to the RML variable `IDOBJ` *before* the rule is applied. This pre-binding of some of the variables is how the tool can manage and schedule the execution of the RML transformation rules. The `IDOBJ` is a value the user of the online webapplication supplies in the form there. If the `obj` elements match, then the other RML variables (`L`, `T`, `O` and `ObjChildren`) are filled with variables from the input `obj`. The `L`, `T` and `O` variables are bound to strings, the `ObjChildren` variable is bound to the children of the `obj` element: a list of elements and all their children. The `consequence` of the rule creates a new `obj` element, using the values bound to the RML variables, and *replaces* the `obj` element in the input with this new `obj` element.

Due to lack of space we restrict the description of the formalization in RML to the rule for the removal of an event from the event-queue, the antecedent is shown in AML notation:

```
km
    classdiagram
        ...
        class name=rml-ClassName
            statemachine
                transition id=rml-IDTRANS
                    trigger
                        op name=rml-TriggerName
                            rml-list name=Params
                ...
    objectdiagram
        ...
        obj class=rml-ClassName id=rml-IDOBJ
                        rml-others=rml-OtherObjAttrs
            queue
                rml-list name=PreEvents
                op name=rml-TriggerName
                rml-list name=PostEvents
```

and this contains some lines with `...` in places where `rml-list` and `rml-use` constructs are used to preserve input context in the output. Here we see that in RML a pattern can be matched that is distributed over remote parts in the XML, the remoteness of the parts is why the rule has so many lines. In short, this rule looks for the name of the trigger that indicates the event that has to be removed from the event-queue, and then simply copies the event-queue without that event. But to find that name of the trigger, a search through the whole km XML model has to take place, involving the following steps.

During application of this rule, the matching algorithm first tries to match the input with the antecedent of the rule, where IDOBJ and IDTRANS are pre-bound RML variables, input to the tool. With these pre-bound variables it can find the correct `obj`, then it finds the `ClassName` for that object. With the `ClassName` the `class` of the object can be found in the `classdiagram` in km XML. When the class of the object is found, the transition in that class with id `TRANSID` can be found and in that `transition` element in the input we can finally find the desired `TriggerName`. The algorithm then looks for an `op` (operation) event with name `TriggerName` in the event-queue of the `obj`, and binds all other events in the event-queue to RML variables `PreEvents` and `PostEvents`. In the consequence of the rule then, all these bound RML variables are available to produce a copy of the input, with the exception that the correct event is removed. As given, the rule removes the first event that matches. It is trivial to change the rule to one that removes only the first event in a queue (by removing the `PreEvents`), or only the last. This is an example that shows that the semantics defined in the RML rules can be easily adapted, even *during* a simulation, and this makes such rules particulary suitable for experimental analysis.

The km application gives comments, for example about the result of the evaluation of a guard of a transition. If the user for instance selects a transition identity that does not correspond with the current state of the object, in the online example if you select (ObjID,TransitionID)=(1,t1) twice, a message is displayed on top of the model, like

```
#   Exiting: Wrong location (object:state_1 transition:start)
```

meaning that transition `t1` can not be taken because the object is in state `state_1` and the transition is defined for a `source` state with name `start`. Such messages do not interfere with the model itself, they are encoded as comments, and the model is unchanged after this message.

The only software a user needs to use the interactive application is a standards compliant browser like Mozilla or Internet Explorer. A user can not only go forward executing a model, but also go *backward* with browser's Back button. This is an example of the benefit of interoperability that XML offers, together with a software architecture and design that is platform independent.

2.4.2 Other examples

If a formalism is expressed in mathematics, then MathML is a generally usable way to express structure, and RML rules extending MathML can capture the dynamics. As an example of this, RML is used for an online interactive theorem prover that can be used to derive proofs for tautologies in propositional logic[1].

Although defining models and semantics in MathML will appeal to the mathematically educated, sometimes it is better to define a new special-purpose XML vocabulary; to make it more concise, better readable, more efficient, and for several other reasons. This was the case in the Archimate [Arc] project where RML has been applied successfully to Enterprise Architecture and Business Models. Rule–based transformations are being used for analysis of models and for visualizations. The RML tutorial in Chapter 3 and the downloadable RML package contain examples in the Archimate language.

2.5 Related work and conclusion

Standards related to RML are XML [XML], MathML [Mat03] and XSLT [XSL]. MathML is a W3C specification for describing mathematics in XML, and it is the problem domain language for the proof example in this paper. XSLT is a W3C language for transforming XML documents into other XML documents, and is discussed in Section 2.2.1. There are also standards that are indirectly connected with XML transformations, like XQuery that can treat XML as a database that can be queried, but a discussion of the many XML standards here is out of scope.

The RuleML community [RUL] is working on a standard for rule-based XML transformations. Their approach differs from the RML approach: RML re-uses the problem domain XML, extended with only a few constructs (in the table in Fig. 2.1) to define rules; whereas RuleML superimposes a special XML vocabulary for rules. This makes the RuleML approach complex and thus difficult to use in certain cases. The idea of using wildcard elements for XML has not been incorporated as such in the RuleML approach, but perhaps it can be added to RuleML and working together with the RuleML community in the future can be interesting.

[1] http://homepages.cwi.nl/˜jacob/MathMLcalc/MathMLcalc.html

There are a number of tools, many of them commercial, that can parse XML and store data in tables like those in a relational database. The user has to define rules for extracting the data, to define what is in the columns and the rows of the tables, to define an entity-relationship model, and other things. Once the data the user is interested in is in the database, a standard query language like SQL can be used to extract data. And then that data can be used to construct new XML. The XML application called XQuery can be used in a similar way, and it is the approach taken by ATL [BDJ$^+$03]. It would be possible to do any transformation with these techniques, but it would be very complex.

The Relational Meta-Language [Pet94] is a language that is also called RML, but intended for compiler generation, which is much more roundabout and certainly not usable for rapid application development like with RML in this paper.

An example of another recent approach is fxt [BS02], which, like RML, defines an XML syntax for transformation rules. Important drawbacks of fxt are that it is rather limited in its default possibilities and relies on hooks to the SML programming language for more elaborate transformations. For using SML a user has to be proficient in using a functional programming language. An important disadvantage of a language like SML is that it is not a mainstream programming language like Python with hundreds of thousands or users worldwide, which makes it unattractive to invest in tools based on SML. The fxt tools are available online but installing them turned out to be problematic.

The experience with several tools as mentioned above leads to the concept of *usability* of a tool in general. Here, a tool is not considered usable enough if it is too difficult to install and configure it and get it to run, or if the most widely used operating system Windows is not supported, or if working with the tool requires a too steep or too high learning curve, for example because the user has to learn a whole new programming language that is not a mainstream programming language. Although the fxt article [BS02] interestingly mentions ”XML transformation ... for non-programmers”, fxt is unfortunately an example of an approach that is not usable enough according to this usability definition.

XML is still gaining momentum and becoming more important and as a result there are many more tools from academic research available, rather too much to mention here as an internet search for “XML tool” reveals hundreds of search results. Unfortunately none of them turned out to be useful in practice for our work according to the above definition of usability, after spending considerable time trying them out.

Other popular academic research topics that could potentially be useful for rule–based XML transformations are term–rewriting systems and systems based on graph grammars for graph reduction. However, the tested available tools for these systems suffer from the same kind of problems as mentioned above: the tools are generally not portable and most will never be portable for technical reasons, and using these tools for XML transformations is an overly complex way of doing things. To use these kind of systems, there has to be first a translation from the problem XML to the special-purpose data structure of the system. And only then, in the tool–specific format, the semantics is defined. But the techniques used in these systems are interesting, especially for very complex or hard transformations, and it looks worthwhile to see how essential concepts of these techniques can be incorporated in RML in the future.

Compared with the related work mentioned above, a distinguishing feature of the RML approach is that RML *re-uses* the language of the problem itself for matching patterns and generating output. This leads in a natural way to a much more usable and clearly defined set of rule–based transformation definitions, and an accompanying set of tools that is being used successfully in practice.

Elements that designate rules	
div	class="rule"
div	class="antecedent" context="yes"
div	class="consequence"

element	attribute	A	C	meaning
Elements that match elements or lists of elements				
rml-tree	name="X"	*		Bind 1 element at this position to RML variable X.
rml-text	name="X"	*		Bind XML text-content to variable X.
rml-list	name="X"	*		Bind a sequence of elements to X.
rml-use	name="X"		*	Output the contents of the RML variable X at this position.
Matching element names or attribute values				
rml-X	...	*		Bind element name to RML variable X.
rml-X	...		*	Use variable X as element name.
...	...="rml-X"	*		Bind attribute value to X.
...	...="rml-X"		*	Use X as attribute value.
...	rml-others="X"	*		Bind *all* attributes that are not already bound to X.
...	rml-others="X"		*	Use X to output attributes.
...	rml-type="or"	*		If this element does not match, try the next element in the antecedent if that also has rml-type=”or”.
Elements that add constraints				
rml-if	child="X"	*		Match if X is already bound to 1 element, and occurs *somewhere* in the current sequence of elements.
rml-if	nochild="X"	*		Match if X does not occur in the current sequence.
rml-if	last="true"	*		Match if the preceding sibling of this element is the last in the current sequence.

A * in the A column means the construct can appear in a rule antecedent. A * in the C column is for the consequence.

Figure 2.1: All the RML constructs

Chapter 3

The RML Tutorial

Author: Joost Jacob

When reading this tutorial you could go directly to Sect. 3.4 **Installing and running** now if you are in a hurry and just want to learn RML quickly. There are examples in Sect. 3.5 **Examples** that will introduce everything incrementally and step by step. You can treat the examples as exercises and try to solve them before looking at the solutions. When trying to solve such exercises you can use Table 2.1 for an overview of all the RML constructs that are defined in the current version (September 23, 2008) of RML. Sections 3.1–3.3 are meant for readers who prefer a little more introduction and explanations.

RML (Rule Markup Language) was designed for ease of use. You do not have to be an experienced programmer to use RML. My experiences with the existing transformation methods for XML were such that I felt it was a good idea to come up with something much more simple and elegant.

It is assumed that the reader does have at least a superficial knowledge of XML, like what are XML element names and attributes and for example the fact that well-formed XML only has one root element. With this tutorial the reader can learn how to transform XML with RML, according to transformation rules that are defined using the input XML itself.

Other approaches for XML transformations do not make much use of the problem domain XML for defining transformation rules. They define the transformations in (complex!) special purpose XML [Cla] or they are more low level, defined in various programming languages.

It is not my goal to replace existing technology for XML transformations, but to add a new, easy to use and interoperable technology. If you have a transformation that is easily done with XSLT for instance, then by all means use XSLT. But sometimes you will need transformations that are hard to program with XSLT, for example removing duplicate child elements, and then RML provides an easy solution for your transformation problem.

Your ideas for improving this tutorial are welcome. I have been trying to make this tutorial easy to understand and readable, but English is not my native language so there are probably lots of grammar and style errors. Please do send your suggestions by email to `Joost.Jacob@gmail.com`.

3.1 The XML vocabulary for the examples

The example XML vocabulary in this tutorial is an XML vocabulary for modeling business processes. Such a vocabulary can be defined with a DTD or with an XML Schema but to save space here Fig. 3.1 gives just an informal definition.

element	attributes	explanation
model		the root element
process	id, name	
role	id, name	
collaboration	id, name	
event	id, name	
object	id, name	
triggering	id, name	must contain a from and to element
realisation	id, name	must contain a from and to element
use	id, name	must contain a from and to element
from	href	
to	href	

Figure 3.1: The XML vocabulary for the examples

The exact meaning of the elements and attributes is not important here, this is left to the imagination of the reader. Also there is no required hierarchy or ordering of elements. Later on in this tutorial, if appropriate, ordering requirements can be assumed and explained for some example.

3.2 How RML works

Details will be explained in Sect. 3.5 but here is already a short description of how RML works.

The simplest RML tool is called **applyrule**. It is a Python[vR95] library that can also be used as as command-line program, and it takes as input some problem domain XML and an RML-rule. Both the XML and the RML-rule will normally be provided in files. The **applyrule** program then transforms the problem XML according to the rule and outputs the result.

3.2.1 Rules

An empty rule looks like shown in Figure 3.2.

```
<div class="rule">
    <div class="antecedent">
        <!-- Insert matching pattern here -->
    </div>

    <div class="consequence">
        <!-- Insert output pattern here -->
    </div>
</div>
```

Figure 3.2: An empty RML-rule

As can be seen in Fig. 3.2 an empty rule consists of div elements like in XHTML. A rule consists of an antecedent (input) and a consequence (output), marked with class attributes of div elements. The XML comments in Fig. 3.2 show what must be done to change this empty rule template to rules that actually do something. How this is done is explained later. The empty rule defines an emptry transformation: the output is the same as the input.

3.2.2 Literal matching

Any XML in the antecedent of a rule that is pure problem domain XML is matched literally. An exception is how the attributes are matched: if the (attributes, value) pairs of an element in the antecedent are a subset of the pairs of an element in the input, and if the elements have the same name, then it is considered a match. This means that input elements can have more attributes that are not involved in the matching process.

3.2.3 Wildcard matching

A core idea of RML is to define XML notation for an XML version of constructs like the `*` and `?` and `+` and others in expressions for text matching with wildcards, as illustrated in Fig. 3.3.

```
d:\tutorial>dir

 Directory of d:\tutorial

11/21/2003  05:14 PM    <DIR>          .
11/21/2003  05:14 PM    <DIR>          ..
11/21/2003  05:11 PM                19 g.bat
11/21/2003  05:07 PM                74 make.bat
11/21/2003  05:07 PM             7,764 tutorial.dvi
11/21/2003  05:07 PM             6,962 tutorial.tex

d:\tutorial>dir tutorial.*

 Directory of d:\tutorial

11/21/2003  05:07 PM             7,764 tutorial.dvi
11/21/2003  05:07 PM             6,962 tutorial.tex
```

Figure 3.3: The * wildcard in action in the Windows XP shell

These wildcard constructs can then be used for matching (parts of) XML. The input that matches a wildcard can then be remembered in RML-variables. The constructs are called *XML wildcard expressions*.

RML uses XML wildcard expressions to bind parts of the input XML into RML-variables. There are XML wildcard *elements* and XML wildcard *attributes*. The XML wildcards elements are special RML elements that can be mixed with the problem domain XML in the antecedent of a rule. The XML wildcard attributes are used for binding attribute values into RML-variables. Exactly how it is done will be made clear later with examples. All the RML constructs are shown in Table 2.1 in the Appendix. As can be seen it all fits in one table, and the RML constructs are designed to be easy to remember. Future versions of RML may add extra constructs, but the ones shown in Table 2.1 are all that were needed so far for the XML transformations I encountered in practice.

3.2.4 Search and replace

The **applyrule** program tries to find the pattern in the antecedent of a rule in the input XML. If it finds a piece of input XML that matches the pattern, then it replaces that piece of input by the consequence of the rule.

3.2.5 The dorules tool

The **dorules** program takes as input some problem XML and a list of RML rules. The RML rules are passed as filenames separated by + characters. For instance:

```
dorules -i myinput.xml -r rule1.xml+rule2.xml
```

It then applies the first rule of the list to the input, just like the **applyrule** program, and if there is a match (the output is different from the input) then the program restarts, taking the generated output and the list of rules as input. If a rule does not match the input then the next rule in the list is tried. If no rule in the list matches the input then the program stops. This program turned out to be useful in practice, alleviating the writing of commandline scripts. Such commandline scripts can perform complex transformations by executing **applyrule** and **dorules** repeatedly with varying rules.

3.2.6 The dorecipe tool

To avoid writing commandline scripts at all, the **dorecipe** program is available. With the **dorecipe** program the user can define a sequence of **applyrule** and **dorules** executions in XML. The XML used is Recipe RML (RRML). An example RRML recipe is shown in Fig. 3.4.

```
<rml-recipe>
    <apply>
        <rule>
            <variable name="ID" value="commandline-ID" />
            <directory name="rules" />
            <filename name="effect1.xml" />
        </rule>
    </apply>
    <iterate>
        <rule>
            <directory name="rules" />
            <filename name="effect3.xml" />
        </rule>
        <rule>
            <directory name="rules" />
            <filename name="effect2.xml" />
        </rule>
    </iterate>
</rml-recipe>
```

Figure 3.4: An RRML recipe

An RRML recipe has a root element called `rml-recipe`. This root element can contain `apply` elements and `iterate` elements. The `apply` element corresponds to the execution of the **applyrule** program and the `iterate` element corresponds to the **dorules** program. An `apply` element must constain exactly 1 `rule` element, an `iterate` element must contain 1 or more `rule` elements. Finally, a `rule` element must contain a `filename` element and it can contain a `directory` element and 0 or more `variable` elements. Later in this tutorial there will be examples showing the usage of recipes.

3.2.7 XML parsing details

RML was designed to transform XML elements and their text content. In the current version of RML only XML elements and their text contents are preserved. This means that for instance XML comments or processing instructions are removed. The reason for this is to make the RML tools as portable as possible, independent of the capabilities of the available XML parsers on a platform. There are many XML tools available outside RML to extract and handle things like processing instructions so I suggest you use those if you need to use XML constructs that are not XML elements. However, contact the author if you have suggestions.

In XML elements produced by the RML tools, the order of attributes in the set of attributes may be changed with respect to the input, this depends on the XML parser on your system that is used by the RML tools. This should not be a problem: relying on attribute order in XML is generally considered bad XML usage.

The RML tools look for the pyRXP module and use that if available. The pyRXP module uses the very fast RXP parser, that is an example of a parser that does not preserve attribute order. If your system is missing the pyRXP parser then the old xmllib Python parser is used, this one is available since Python version 1.5.2 from 1999. Support for more parsers may be added in the future.

3.3 Future versions of RML

Some ideas for future versions of RML that would be simple to implement but were not necessary in the XML languages in the projects it was used for are listed here.

- String concatenation.

  Example: `<newprefix+rml-X />` in the consequence. If `X` is bound (in the antecedent) to ”MyName” then this will produce a `<newprefixMyName />` element.

- Arithmetic

  Example: `<...   ...= 10+rml-X />` in the consequence. If `X` is bound to a string representing a number, then the sum of 10 and this number becomes the attribute value.

- XPath support.

  Example: `<rml-if xpath="model/*/process" />` will let a match only succeed if the previous element is a `processs` element with a `model` ancestor.

3.4 Installing and running

Unzip the supplied zipfile, this will create a directory called `rml`, containing several other subdirectories. The RML tools, **applyrule** and **dorules** and **dorecipe**, are in the `rml` directory. It is assumed the reader does know how to run commandline programs and how to go to directories on his or her operating system. If you have a default Python [vR95] installation on a Windows computer then you can call Python files as executable programs, because the `.py` filename extension will be marked as executable. To get Python, go to http://www.python.org and download the executable installer. In the example commands you can then use

```
C:\mystuff\rml\applyrule.py -i myinput.xml -r myrule.xml
```

from any other directory, assuming you did install RML in `C:\mystuff`. With linux or unix systems you usually have to prepend `python` to commands, for instance

```
$ python applyrule.py -i myinput.xml -r myrule.xml
```

If one of the tool programs is run without arguments then a short usage help is output.

The tutorial examples are in subdirectory `examples` below `tutorial` below `rml`. If you go to the examples directory you can run the **applyrule** program on file ”input.xml” there using the empty rule in file ”rule.empty.xml” by issuing the

```
..\..\applyrule.py -i input.xml -r rule.empty.xml
```

command. This will output the contents of the file ”input.xml”, since nothing was matched. The output is pretty printed, with more indentation for elements deeper down the tree hierarchy, and with attributes indented from the element name and below each other. Figure 3.5 shows an input and the output from applying the empty rule.

Input:

```
<model xmlns="Concepts.ArchiMate.Generic"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="Concepts Generic.xsd">

  <event id="008" name="request for insurance"/>
  <process id="009" name="investigate"/>

  <triggering id="015" name="triggers"><from href="008"/>
      <to href="009"/></triggering>
</model>
```

Output:

```
 <model
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns="Concepts.ArchiMate.Generic"
     xsi:schemaLocation="Concepts Generic.xsd" >
    <event
       name="request for insurance"
       id="008" />
    <process
       name="investigate"
       id="009" />
    <triggering
       name="triggers"
       id="015" >
      <from
         href="008" />
      <to
         href="009" />

    </triggering>

  </model>
```

Figure 3.5: Input and output when using the empty rule

3.5 Examples

3.5.1 Deleting an element

Deleting an element using a literal match

Example 001 Suppose you want to transform

```
<model>
    <event name="request for insurance" id="008" />
    <process name="investigate" id="009" />
    <process name="formalize request" id="010" />
</model>
```

availabe in file input.001.xml

to

```
 <model>
    <event name="request for insurance" id="008" />
    <process name="formalize request" id="010" />
 </model>
```

, removing the `process` element with name `investigate`. This can be done with the RML rule

```
<div class="rule">
    <div class="antecedent">
        <process name="investigate" id="009" />
    </div>
    <div class="consequence" />
</div>
```

rule.001.xml

If you have the input in file `input.001.xml` and the rule in file `rule.001.xml` then you can do this transformation on the commandline with the command

```
applyrule --input input.001.xml --rule rule.001.xml
```

The files for the examples, in this case `input.001.xml` and `rule.001.xml`, are in your `examples` directory.

How does RML apply this rule to the input? What happens is that the **applyrule** program looks in the `antecedent` of the rule and finds the `process` element there. This is a literal element, there are no special RML features in the element. The program then loads the input in its memory and searches the input for such an element. In our case it finds such an element and then replaces it by the contents of the `consequence` of the rule (in this case nothing). The program then outputs the modified input. If the program would not have found a matching element then the output would be the same as the input, but "pretty-printed" (see Fig. 3.5).

Output is printed to screen, there are no special features to produce files. The normal I/O redirection of the operating system can be used to produce files, for instance

```
applyrule --input input.xml --rule rule.xml > myoutput.xml
```

Deleting an element with a specific attribute

Example 002 Now suppose you want to remove a `process` element from `input.001.xml` and you do know it has a `name="investigate"` and an `id` attribute but you don't know the value of the `id` attribute. This RML rule does what you want:

```
<div class="rule">
    <div class="antecedent">
        <process name="investigate" id="rml-X" />
    </div>
    <div class="consequence" />
</div>
```

rule.002.xml

This is an RML rule with a special RML feature in it: the `"rml-X"` attribute value of the `id` attribute. In RML, if an attribute value starts with `rml-` then it is considered an RML variable. The name of the variable is what comes after the `rml-`. If you use this rule, what happens is that the **applyrule** program does the same thing as in example 001, but instead of looking for an `id="009"` it now only looks for an `id` attribute and it puts the value of the attribute it finds into RML variable `X`. This variable `X` is not used anywhere else in the rule so for the rest of **applyrule**'s program execution it is just ignored.

But what if there are 2 `process` elements with `name="investigate"` and an `id` attribute? Then the first one in the input XML will be removed, the first being the first one encountered when reading the XML input file from top to bottom.

Example 002 can reuse the file `input.001.xml` so there is no need for a `input.002.xml` file, but there is a `rule.002.xml` in your `examples` directory.

Deleting an element with a specific attribute value

Example 003 The following rule removes a `process` element if it has an `id` attribute with value `009`, ignoring all other attributes and values.

```
<div class="rule">
    <div class="antecedent">
        <process id="009" />
    </div>
    <div class="consequence" />
</div>
```

rule.003.xml

This rule works because RML does subset matching: if the element name and all the attributes of a pattern element match, then it is considered a match even if the matching input element has more attributes.

If `<process id="009" />` is replaced by `<process id="rml-X" />` then even the attribute value does not matter: the first `process` element with an `id` attribute is removed.

3.5.2 Changing an element

So far we did see only rules that delete elements. The emphasis was on how to match input elements, how to select elements that are to be deleted. In this Section we will see how element names and attributes can be changed into something else. This will also explain more about RML-variables.

Changing an element name

Example 004 Suppose you want to change the name of the `<process name="investigate" id="009" />` element to `MyProcess`. The rule in file `rule.004.xml` does just that.

```
<div class="rule">
    <div class="antecedent">
        <process name="investigate" id="009" />
    </div>
    <div class="consequence">
        <MyProcess name="investigate" id="009" />
    </div>
</div>
```

rule.004.xml

But this rule only works for a `process` with attributes `name="investigate"` and `id="009"`.

Changing an element name and copying all attributes

Example 005 What if you just want to change the element name like in Example 004 and copy all the other attributes. This is useful for instance if you don't know all the attributes. Then you need the special RML attribute `rml-others`. It puts all attributes that do not appear elsewhere in the element into an RML variable. An RML variable is denoted by a leading `rml-`. Rule `rule.005.xml` shows the solution.

```
<div class="rule">
    <div class="antecedent">
        <process rml-others="rml-X" />
    </div>
    <div class="consequence">
        <MyProcess rml-others="rml-X" />
    </div>
</div>
```

rule.005.xml

Changing all `process` element names: Iterating a rule

Example 006 But `rule.005.xml` only changes 1 element. If you want to change all `process` elements to `MyProcess` elements then you could just repeatedly use **applyrule** until there are no more `process` elements left. But you can also use the `dorules` program here. The `dorules` program has a

`--rules` parameter instead of a `--rule` parameter, it takes a set of rules as the value of that parameter, where rules are separated by a `+` character. In this example there is only one rule. Run
`dorules -i input.002.xml -r rule.005.xml` for the desired effect.

Changing attribute values

Example 007 Example 006 did bind a set of attributes to an RML variable. We can also bind an attribute value to an RML variable. We did that already in Example 002, but here we will also use the value of the attribute in the output.

Rule `rule.007.xml` shows how to search for a `process` with `name="investigate"` and to change it to a `process` with `name="SomethingElse"`. The `id` attribute with its value is copied to the output. This works even when you don't know the value of the `id` attribute, the RML variable with name `A` is used for that.

```
<div class="rule">
    <div class="antecedent">
        <process name="investigate" id="rml-A" />
    </div>
    <div class="consequence">
        <process name="SomethingElse" id="rml-A" />
    </div>
</div>
```

rule.007.xml

3.5.3 RML variables for elements

Section 3.5.2 introduced RML variables. This Section will say more about RML variables. The antecedent of an RML rule contains XML from the problem domain XML vocabulary, mixed with other RML constructs. Together they form a matching pattern. When the rule is applied, this matching pattern is matched against the XML input. When a match occurs, the RML variables in the antecedent are filled with values. The type of these values can be:

- a string (element name, or attribute value),

- a set of attributes and their values,
- one XML element,
- a list of XML elements from the problem domain.

The last two are introduced next.

RML variables for lists of elements and their children

Example 008 Duplicate all childs of a `model` element.

```
<div class="rule">
    <div class="antecedent" >
        <model>
            <rml-list name="A" />
        </model>
    </div>
    <div class="consequence">
        <model>
            <rml-use name="A" />
            <rml-use name="A" />
        </model>
    </div>
</div>
```

rule.008.xml

The `rml-list` RML element stores a list of elements *at that position* in the pattern into an RML variable. The RML variable can be output in the consequence of a rule with the `rml-use` RML element. All children elements of elements in the list will also be copied.

RML variables for complete elements and their children

Example 009 Duplicate the first child of a `model` element.

```
<div class="rule">
    <div class="antecedent" >
        <model>
            <rml-tree name="A" />
            <rml-list name="B" />
        </model>
    </div>
    <div class="consequence">
        <model>
            <rml-use name="A" />
            <rml-use name="A" />
            <rml-use name="B" />
        </model>
    </div>
</div>
```

rule.009.xml

The `rml-tree` element matches only 1 element (and all its possible children), the `rml-list` element after it in the antecedent of the rule matches the rest of the elements in that list.

Defining RML variables for elements or lists of element with `rml-bind`

New in RML 1.6.

You can bind RML variables for elements (or for lists of elements) in the matching process, but you can also define them “manually” with rml-bind:

```
<div class="rule">
    <div class="antecedent" >
        <model>
            <rml-bind name="A">
                <MyNewElement />
            </rml-bind>
            <rml-list name="B" />
        </model>
    </div>
    <div class="consequence">
        <model>
            <rml-use name="A" />
        </model>
    </div>
</div>
```

rule.009a.xml

This rule replaces all children of `model` with the `MyNewElement` element. Here `<MyNewElement/>` is bound to RML variable `A`. Instead of only `<MyNewElement/>` you can also put a list of elements there, and you can use RML variables to define the elements (for example RML variables for element names and attribute values). The rule.009a.xml is equivalent with a rule that has no rml-bind element and just has `<MyNewElement/>` in the consequent of the rule in place of the `rml-use` element there. So why bother with `rml-bind`? In Section 3.5.5 in example 011a we will see how manually binding with `rml-bind` can be useful.

3.5.4 RML variables for text content

The `<rml-text name="SomeName" />` construct is used to bind XML text-content. It is used in the same way as the `rml-tree` element, but it will only match if there is XML text-content to be found in the matching position of the input. So you can not match an *element* and put it in the `SomeName` variable, if you want that, then you have to use `rml-tree`. You can use the `SomeName` RML variable just like any other RML variable; use it like `rml-SomeName` for an element name in the output or for an attribute name or attribute value, or use it like `<rml-use name="SomeName" />` for text-content of an XML element.

3.5.5 Adding constraints with `rml-if`

Example 010 Transform

```
<model>
    <collaboration id="004" name="Negotiation">
        <role id="001" name="Intermediary"/>
        <role id="002" name="Customer"/>
    </collaboration>
    <process id="009" name="investigate"/>
    <role id="002" name="Customer"/>
    <process id="010" name="formalize request" />
</model>
```

input.010.xml

into

```
<model>
    <collaboration id="004" name="Negotiation">
        <role id="001" name="Intermediary"/>
        <role id="002" name="Customer"/>
    </collaboration>
    <process id="009" name="investigate"/>
    <role id="002" name="Customer"/>
    <process id="010" name="formalize request" />
    <role id="001" name="Intermediary"/>
</model>
```

by executing this rule:

```
<div class="rule">
    <div class="antecedent" >
        <model>
            <collaboration rml-others="CollAttrs">
                <rml-tree name="A" />
                <rml-tree name="B" />
            </collaboration>
            <rml-list name="L1" />
            <rml-if child="B" />
            <rml-list name="L2" />
        </model>
    </div>
    <div class="consequence">
        <model>
            <collaboration rml-others="CollAttrs">
                <rml-use name="A" />
                <rml-use name="B" />
            </collaboration>
            <rml-use name="L1" />
            <rml-use name="L2" />
            <rml-use name="A" />
        </model>
    </div>
</div>
```

rule.010.xml

In the example the `Intermediary` is added as a child of `model`. The rule looks for a `model` element with a `collaboration` child that in turn has exactly 2 children elements, where the second child elements is also a child of the `model` element. If that is the case, the rule adds the first child to the children of `model`.

The `rml-if` elements are elements that constrain whether a match succeeds or not. The `rml-if child="SomeVar"` element only succeeds if the element bound to `SomeVar` appears somewhere in the current list of elements. `SomeVar` has to be bound earlier in the rule with an `rml-tree name="SomeVar"` element. There is also a `rml-if nochild=X />` element that succeeds only if `X` does *not* appear in the current list. These 2 constraint adding elements are more complex than I would like, but I have found good usage for them in practice.

Example 011 If you repeat `rule.010.xml`, using the output as input, then you add `Intermediary` elements every time. To prevent that, rewrite the rule as:

```
<div class="rule">
    <div class="antecedent" >
        <model>
            <collaboration rml-others="CollAttrs">
                <rml-tree name="A" />
                <rml-tree name="B" />
            </collaboration>
            <rml-list name="L1" />
            <rml-if child="B" />
            <rml-if nochild="A" />
            <rml-list name="L2" />
        </model>
    </div>
    <div class="consequence">
        <model>
            <collaboration rml-others="CollAttrs">
                <rml-use name="A" />
                <rml-use name="B" />
            </collaboration>
            <rml-use name="L1" />
            <rml-use name="L2" />
            <rml-use name="A" />
        </model>
    </div>
</div>
```

rule.011.xml

The only difference is the line with `<rml-if nochild="A" />` in the antecedent of the rule. If I execute `..\..\dorules.py -i input.010.xml -r rule.010.xml` on my computer, then the program *hangs* until the system runs out of memory: It tries to add an infinite number of `Intermediary` elements and I have to use the Break key to stop it. But if I use `dorules` with `rule.011.xml` then it stops with the desired effect. It stops because the second time it tries to apply the rule, the match fails because `nochild` fails because there *is* an `Intermediary`. The `dorules` program stops when it can not change the input anymore with any of its rules.

Example 011a In example 011 we were able to stop iteration by simple adding a `<rml-if nochild="A">` element. The value for the `A` variable was found earlier in the rule. But sometimes we can not do this, especially when the value that we want to bind to the variable is not present in the input. An example is when we want to create a completely new element and bind it to an RML variable such that we we can use the RML variable in `rml-if` constructs.

Suppose we want to transform input.011a.xml:

```
<model>
    <collaboration id="004" name="Negotiation">
        <role id="001" name="Intermediary"/>
        <role id="002" name="Customer"/>
    </collaboration>
</model>
```

input.011a.xml

into output.011a.xml:

```
<model>
    <collaboration id="004" name="Negotiation">
        <role id="001" name="Intermediary"/>
        <role id="002" name="Customer"/>
    </collaboration>
    <Intermediary id="001"/>
</model>
```

output.011a.xml

creating a new element with name Intermediary. Then we can use rule.011a.xml:

```
<div class="rule">
    <div class="antecedent" >
        <model>
            <collaboration rml-others="CollAttrs">
                <rml-list name="PreRoles"/>
                <role id="rml-idA" name="rml-A" />
                <rml-list name="PostRoles"/>
            </collaboration>
            <rml-list name="Tail"/>
        </model>
    </div>
    <div class="consequence">
        <model>
            <collaboration rml-others="CollAttrs">
                <rml-use name="PreRoles"/>
                <role id="rml-idA" name="rml-A" />
                <rml-use name="PostRoles"/>
            </collaboration>
            <rml-use name="Tail"/>
            <rml-A id="rml-idA" />
        </model>
    </div>
</div>
```

rule.011a.xml

But when we apply rule.011a.xml to the result (in output.011a.xml) again, it produces *another* Intermediary element. We want to stop that, and instead add a Customer element, like in output.011b.xml:

```
<model>
  <collaboration id="004" name="Negotiation">
    <role id="001" name="Intermediary"/>
    <role id="002" name="Customer"/>
  </collaboration>
  <Intermediary id="001"/>
  <Customer id="002"/>
</model>
```

output.011b.xml

To achieve this, use rule.011b.xml:

```
<div class="rule">
    <div class="antecedent" >
        <model>
            <collaboration rml-others="CollAttrs">
                <rml-list name="PreRoles"/>
                <role id="rml-idA" name="rml-A" />
                <rml-list name="PostRoles"/>
            </collaboration>
            <rml-bind name="New">
                <rml-A id="rml-idA" />
            </rml-bind>
            <rml-if nochild="New"/>
            <rml-list name="Tail"/>
        </model>
    </div>
    <div class="consequence">
        <model>
            <collaboration rml-others="CollAttrs">
                <rml-use name="PreRoles"/>
                <role id="rml-idA" name="rml-A" />
                <rml-use name="PostRoles"/>
            </collaboration>
            <rml-use name="Tail"/>
            <rml-A id="rml-idA" />
        </model>
    </div>
</div>
```

rule.011b.xml

When we apply rule.011b.xml iteratively until the output is stable (with the dorules tool: ”dorules.py -i input.011a.xml -r rule.011b.xml”) we get the desired output.011b.xml. The rule works by binding the desired new element to an RML variable with the name `New`, and using this `New` variable in the `<rml-if nochild="New"/>` test; preventing a match if the new element is already there.

Example 012 Suppose you want to output the second `role` in the first `collaboration` in `input.010.xml`, producing

```
<role id="002" name="Customer"/>
```

This rule:

```
<div class="rule">
    <div class="antecedent">
        <model>
            <collaboration>
                <role/>
                <rml-tree name="A"/>
            </collaboration>
        </model>
    </div>

    <div class="consequence">
        <rml-use name="A" />
    </div>
</div>
```

rule.012a.xml

produces the desired output. And when applying that rule to

```
<model>
    <collaboration id="004" name="Negotiation">
        <role id="001" name="Intermediary"/>
        <role id="002" name="Customer"/>
        <role id="003" name="Office"/>
    </collaboration>
    <process id="009" name="investigate"/>
    <role id="002" name="Customer"/>
    <process id="010" name="formalize request" />
</model>
```

input.012.xml

it also outputs the `Customer`. But what if you want the rule only to work if there are exactly 2 child elements in the `collaboration`? Then use rule:

```
<div class="rule">
    <div class="antecedent">
        <model>
            <collaboration>
                <role/>
                <rml-tree name="A"/>
                <rml-if last="true"/>
            </collaboration>
        </model>
    </div>

    <div class="consequence">
        <rml-use name="A" />
    </div>
</div>
```

rule.012.xml

The `<rml-if last="true"/>` element makes the matching of the rule only succeed if the previous element is the last in a list. If you apply `rule.012.xml` to `input.012.xml` then you get the contents of the input back: no match. But `rule.012.xml` *does* work on `input.010.xml`.

3.5.6 Match choice with `rml-type="or"`

Example 013 If you want to match

```
<model>
    <role id="001" name="Intermediary"/>
</model>
```

input.013a.xml

or

```
<model>
    <role id="002" name="Customer"/>
</model>
```

input.013b.xml

but not

```
<model>
    <role id="003" name="Insurance Company"/>
</model>
```

input.013c.xml

then you can use rule `rule.13.xml`:

```
<div class="rule">
    <div class="antecedent">
        <model>
            <role id="001" name="Intermediary"
                  rml-type="or" />
            <role id="002" name="Customer"
                  rml-type="or" />
        </model>
    </div>

    <div class="consequence">
        <matched how="allright" />
    </div>
</div>
```

rule.13.xml

With the special attribute `rml-type="or"`, the RML tools try the next element if a match fails on an element, but only if that next element also has this special attribute.

3.5.7 How to remove duplicate siblings

Example 014 Some of the inspiration leading to RML came when I had to transform something like

```
<model>
    <role id="003" name="Insurance Company"/>
    <whatever />
    <role id="003" name="Insurance Company"/>
    <evenmore />
</model>
```

input.014.xml

to

```
<model>
    <role id="003" name="Insurance Company"/>
    <whatever />
    <evenmore />
</model>
```

, just removing duplicate siblings. This looks simple but I found out it was very hard to do with for example XSLT. The RML rule for this is not difficult:

```
<div class="rule">
    <div class="antecedent">
        <model>
            <rml-list name="list1" />
            <role rml-others="A" />
            <rml-list name="list2" />
            <role rml-others="A" />
            <rml-list name="list3" />
        </model>
    </div>

    <div class="consequence">
        <model>
            <rml-list name="list1" />
            <role rml-others="A" />
            <rml-list name="list2" />
            <rml-list name="list3" />
        </model>
    </div>
</div>
```

rule.014.xml

This rule also shows a typical usage of `rml-list` elements: all the elements, and their children, around the elements you are interested in, are remembered in variables (here `list1` and `list2` and `list3`. With this pattern you can "preserve the context" of the elements you want to match. Example 016 also shows this pattern.

Example 015 And for Example 014 even this rule works:

```
<div class="rule">
    <div class="antecedent">
        <model>
            <rml-list name="list1" />
            <rml-tree name="A" />
            <rml-list name="list2" />
            <rml-use name="A" />
            <rml-list name="list3" />
        </model>
    </div>

    <div class="consequence">
        <model>
            <rml-use name="list1" />
            <rml-use name="A" />
            <rml-use name="list2" />
            <rml-use name="list3" />
        </model>
    </div>
</div>
```

rule.015.xml

, not only for duplicate `role` elements, but for *any* duplicate elements in the `model`. This rule makes use of the fact that variable `A` has been bound in the line with `<rml-tree name="A" />`, and then a `<rml-use name="A" />` is allowed not only in the consequence, but also in the antecedent of a rule.

3.5.8 Iterating sets of rules

The `dorules` tool accepts a list of rulefilenames in parameter `--rules` (or`-r`), instead of just one rule like the `applyrule` tool.

It then applies the first rule of the list to the input, just like the **applyrule** program, and if the rule can be successfully applied (the output is different from the input) then the program restarts, taking the generated output and the list of rules as input. If a rule does not match the input then the next rule in the list is tried. If no rule in the list matches the input then the program stops.

Iterating sets of rules is often useful. A typical usage pattern is when you want to create new elements with data from 2 original elements, but you don't know the order of the 2. Then you write 2 rules and let `dorules` handle it.

Example 016 Transform

```
<model>
    <triggering id="016" name="triggers">
        <from href="009"/>
        <to href="010"/>
    </triggering>
    <triggering id="015" name="triggers">
        <from href="008"/>
        <to href="009"/>
    </triggering>
</model>
```

input.016.xml

into

```
<model>
    <triggering id="016" name="triggers">
        <from href="008"/>
        <to href="010"/>
    </triggering>
</model>
```

by executing `dorules` with this rule:

```
<div class="rule">
    <div class="antecedent">
        <model rml-others="modelAttrs">
            <rml-list name="Prelude" />
            <rml-R1 name="rml-N1">
                <from href="rml-F1"/>
                <to href="rml-ID" />
            </rml-R1>
            <rml-list name="Between" />
            <rml-R1 name="rml-N1" rml-others="AttrRest">
                <from href="rml-ID"/>
                <to href="rml-T1" />
            </rml-R1>
            <rml-list name="Epilog" />
        </model>
    </div>

    <div class="consequence">
        <model rml-others="modelAttrs">
            <rml-use name="Prelude" />
            <rml-R1 name="rml-N1" rml-others="AttrRest">
                <from href="rml-F1"/>
                <to href="rml-T1" />
            </rml-R1>
            <rml-use name="Between" />
            <rml-use name="Epilog" />
        </model>
    </div>
</div>
```

rule.016a.xml

and this rule:

```
<div class="rule">
    <div class="antecedent">
        <model rml-others="modelAttrs">
            <rml-list name="Prelude" />
            <rml-R1 name="rml-N1" rml-others="AttrRest">
                <from href="rml-ID"/>
                <to href="rml-T1" />
            </rml-R1>
            <rml-list name="Between" />
            <rml-R1 name="rml-N1">
                <from href="rml-F1"/>
                <to href="rml-ID" />
            </rml-R1>
            <rml-list name="Epilog" />
        </model>
    </div>

    <div class="consequence">
        <model rml-others="modelAttrs">
            <rml-use name="Prelude" />
            <rml-R1 name="rml-N1" rml-others="AttrRest">
                <from href="rml-F1"/>
                <to href="rml-T1" />
            </rml-R1>
            <rml-use name="Between" />
            <rml-use name="Epilog" />
        </model>
    </div>
</div>
```

rule.016b.xml

issuing the command:
`dorules -i input.016.xml -r rule.016a.xml+rule.016b.xml`

Note that the order of the rules in the list is significant. The order is not important in this example but in another it could be. This example is also another example of “context preserving” with `rml-list` elements, storing the context in `Prelude`, `Between` and `Epilog`.

3.5.9 Turning a list into a hierarchy

Example 017 Transform

```
<model>
    <role id="001" name="Intermediary"/>
    <collaboration id="004" name="Negotiation"/>
    <process id="009" name="investigate"/>
    <process id="010" name="formalize request"/>
</model>
```

input.017.xml

into

```
<model>
    <role id="001" name="Intermediary" >
        <collaboration >
          <process id="009" name="investigate" >
            <process id="010" name="formalize request"/>
          </process>
        </collaboration>
    </role>
</model>
```

by executing `dorules` with this rule:

```
<div class="rule">
    <div class="antecedent" >
        <rml-Top>
            <rml-Name rml-others="A" />
            <rml-Name2 rml-others="B" />
            <rml-list name="L" />
        </rml-Top>
    </div>
    <div class="consequence">
        <rml-Top>
            <rml-Name rml-others="A">
                <rml-Name2 rml-others="B">
                    <rml-use name="L" />
                </rml-Name2>
            </rml-Name>
        </rml-Top>
    </div>
</div>
```

rule.017.xml

This rule ”deepens” all lists in the input when applied iteratively wih `dorules`.

3.5.10 Pre-binding string variables on the command-line

Example 018 If you have this input:

```
<model>
    <event name="request for insurance" id="008" />
    <process name="investigate" id="009" />
    <process name="investigate" id="010" />
</model>
```

availabe in file input.018.xml

and you would like to remove the `process` with `id="010"`, then you need somthing very similar to the `rule.002.xml` you wrote earlier. But that rule removes the `id="009"` process, because that is the first that matches the antecedent of the rule (`<process name="investigate" id="rml-X" />`).

A solution is to bind commandline variables to RML variables. If you issue the command:
`applyrule -i input.018.xm -r rule.002.xml X=010` then the `id="010"` will be removed.

What happens is that the *string* `010` is bound to RML variable `X`, meaning that the RML tools now treat "rml-X" in a rule as "010".

3.5.11 Using recipes

RML recipes are stated in Recipe RML (RRML), an XML vocabulary for RML recipes. With RML recipes the user can define sequences of `applyrule` and `dorules` executions. See Sect. 3.2.6 and Fig. 3.4, there is an example recipe. There are plans for a future version of RRML with `rule` elements that can execute XSLT transformations too. Or that can execute arbitrary programs...let me know what you would like.

I hope you enjoyed this tutorial. Good luck with your XML transformations!

Part II

Component Models and Analysis

Chapter 4

The OMEGA Component Model

Author: Joost Jacob

4.1 Introduction

In this paper we introduce a formal model of components as developed in the IST project OMEGA (IST-2001-33522, [OME]) sponsored by the European Commission. The aim of this project is the definition of a development methodology in UML for embedded and real-time systems based on formal techniques. The approach followed in OMEGA is based on a formal semantics of a suitable subset of UML 1.4 which includes class and state diagrams, Object Constraint Language (OCL), use case diagrams, and Live Sequence Charts ([DH01], an extension of UML's sequence diagrams). Some of the OMEGA members have been involved in the design of a component model for UML 2.0 that will be finished in the course of 2004 and the OMEGA component model has influenced the UML 2.0 component model; therefore the approach presented in this paper is compatible with the approach taken in UML 2.0.

The main rationale of our component model is to extend the above subset of UML as used in OMEGA with additional structuring and abstraction mechanisms which allow a modeling discipline and the application of formal techniques based on "interfaces". The basic idea of a component presented in this paper is that of a high-level software abstraction like a module which encapsulates its internal structure and which provides an interface specifying the exported (also called provided) and imported (also called required) operations and signals. Components can be hierarchically composed from basic components and relations between provided and required interfaces. Basic components are defined as sets of classes together with the provided and required interfaces. Components interact via Ports. In our model a Port is an object realizing an interface and/or depending on an interface of another component. Ports, like any other object-instances, can be created dynamically. In this sense our notion of a Port differs from the usual UML definition of an interface. Since we view components as a software abstraction, components themselves cannot be instantiated but only its Ports are instantiated. If there is only one Port instantiated for a component then this Port can be regarded as "the component instance" or "the component" and this phrase is sometimes used to make text more readable.

We show how our component model provides a general framework for the classification of and relationships between the UML concepts mentioned above as used in OMEGA, by adding *component diagrams* and *architectural diagrams*. Architectural diagrams are used to describe certain run–time properties of components which are independent of the actual deployment on a certain platform. There is an analogy between component diagrams and class diagrams and likewise between architectural diagrams and object diagrams, and this analogy can be used to design our new diagrams using CASE tools that do not support the new component model diagrams yet. Finally, we discuss the possible usage of the OMEGA component model for verification purposes.

The first version of our component model was presented as an OMEGA milestone document in June 2002.

4.2 The Component Model

In this section, we present a meta–model for our notion of components. In this meta–model we extend standard UML entities, the building blocks, like class and interface. Since we only use a few UML entities it will be easy in the future to make the meta–model compatible with UML 2.0 [OMG] once that has reached a stable version, and to fit it in with the new MOF [MOF] version that is under development. To avoid confusion with existing UML entities, in the rest of this paper we will use a capital for the first letters of the names of entities that are our extensions to UML.

Component Models as high–level class diagrams

Our starting point is a model of components which provides a high-level software abstraction like that of a *module* which encapsulates its internal structure and which provides an interface specifying the exported (also called provided) and imported (also called required) operations and signals (as defined by the OMEGA kernel model language in [OME] and [DJPV03]. The interface of a component is structured into *Component Interfaces.* Component Interfaces are like ordinary UML interfaces but they have to adhere to the usage rules for Component Interfaces we specify below in this section. A Component Interface consists of a collection of signatures of operations and signals, but contrary to ordinary UML interfaces Component Interfaces do not contain attributes. In comparison with UML diagrams, a component model is similar to a class diagram. Later, in Sect. 4.4, we will introduce diagrams for components, so-called architectural diagrams, that are similar to UML object diagrams.

The underlying class diagram

In an OO setting there is always a class diagram underlying an application. The same is true for a component based application designed with our component model. In the OMEGA deliverable D1.1.2 we have presented a formal reduction from a hierachical component model to a flat class diagram. In this paper we will present in Sect. 4.5 a formal justification of our component model in terms of a compositional trace semantics and its corresponding logics.

Introducing Ports

Component Interfaces are grouped into Ports. Component Ports correspond with special purpose classes inside components that provide the only interaction points between components. At runtime, all communication between components is going via instantiated Ports. In our component model, a Port is used as a class, and it is also used as a type specification for one or more runtime objects. Ultimately these runtime objects are instances of classes in the underlying class diagram, because our model is designed in an OO setting.

Why is an object-oriented component model useful?

The underlying class diagram can possibly be huge; this is one place where a component model can be useful because one component can abstract from many classes. Also, it is possible to design a component with a Port, and to be specific about the services the Port requires and provides, without having to specify already exactly what class will used for instantiating the Port; this supports better top–down design methodologies. Our component diagram groups classes in an underlying class diagram into components, and it groups associations in that class diagram into Ports and Component Interfaces and the associations and connections between them. As such it provides a high-level view of a class-based application which is both suited for top-down design and compositional analysis.

UML 2.0

Syntactically the components in our component model are much like the components in the UML 2.0 submission by U2Partners in September 2002 and in Januari 2003. One of our main objectives in OMEGA is the development of an OMEGA component model which is compatible with their UML 2.0 submissions. But there are some semantic differences that will appear in the rest of this paper. We can mention here already one of the most important differences: in the submissions by U2Partners a component *itself* is instantiable whereas in our model it is the Component *Ports* that are instantiated (as instances of UML classes); this way the component provides a conservative extension of the underlying object-orientation so it can remain a software abstraction. Another difference is that in order to keep our model small, simple and elegant, we do not explicitly model connectors and therefore we have not defined new UML entities for connectors. This provides a user of our component model with a choice: the user may decide to extend our model and use the UML 2.0 connectors, or the user can choose to model connectors as components themselves.

4.2.1 Blackbox Components

A Blackbox Component gives a *blackbox view* of a component in a blackbox diagram. Inside a Blackbox Component nothing is visible, only the Interfaces of the component that are to be used in a design outside of the component are visible, and the grouping of these Interfaces into Ports is visible.

The meta–model for a Blackbox Component is contained in Fig. 4.1; the Basic Component and UML Class and Component System boxes and their relations do not belong to it but will be introduced later. As can be seen in the figure, we have modeled a Blackbox Component as a specialization of a UML Classifier. In a future MOF version a component could well be better modeled specializing another (future?) MOF construct that is more suitable for our purposes, or perhaps more than one construct. For now we are basing our meta–models on UML 1.4 and therefore we use the Classifier.

The same goes for the other specializations from UML entities we use in the figure, again for now we make do with UML 1.4 entitities. Adapting the meta–model to the next MOF or UML version should pose no problems that cannot be overcome easily.

In Fig. 4.1 we can see that Blackbox Components can have several Ports, and a Port can have several Provided Component Interfaces and several Required Component Interfaces. The other way around, a Component Interface is associated with one Port, and a Port belongs to one Blackbox Component.

A Blackbox Component is drawn as a box, optionally with the UML 1.x component symbol in a corner to make it extra clear that the box is a component. Ports are drawn as small squares on the edges of a component box. In the blackbox view the association between a Port and Component Interfaces can be shown with the “lollypop” notation, or with UML dependency and UML realization associations to expanded interfaces (boxes with the name of the interface and a stereotype indiciation and a list of services). This is in accordance with standard UML 1.x notation; an example with the two notations is shown in Fig. 4.2. We therefore propose to extend the Kernel Model Language with UML realization associations. Note that these associations do not affect the semantics, i.e., they encode only static information which can be checked by a preprocessor.

There can be UML dependency relations from Provided Component Interfaces to Required Component Interfaces on the same component. This means that if a user wants to use one of the services of the Provided Interface, the Required Interface must be realized, or else the service is not

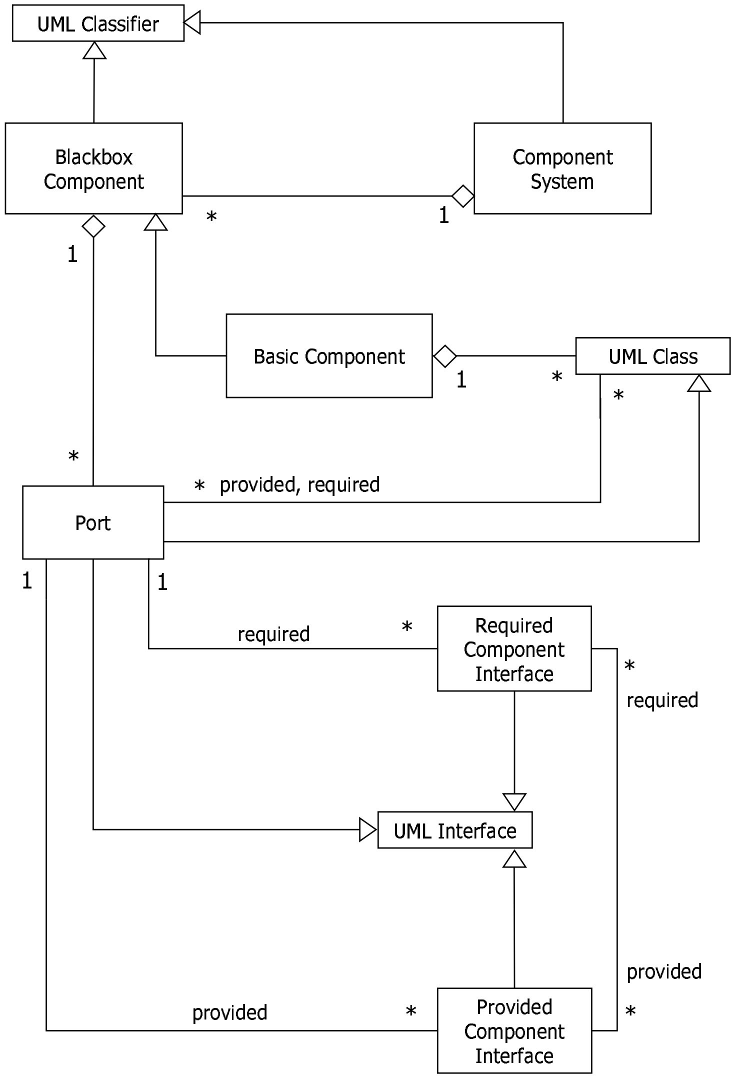


Figure 4.1: The combined UML Meta-Model for our component models

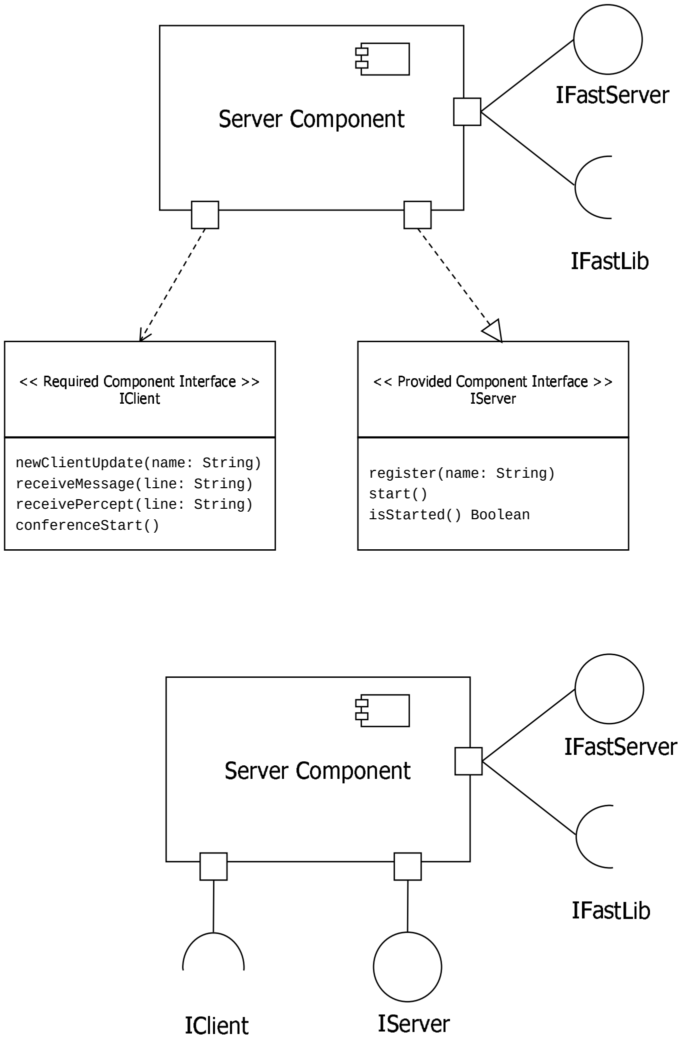


Figure 4.2: A Blackbox Component; top: with 2 expanded interfaces, bottom: with all interfaces in elided form.

guaranteed. Note that we see here a coupling between a Provided and a Required Interface on the *same* component. This special dependency expresses the fact that the same object that provides the services in the Provided Interface, depends on the services in the Required Interface. Such an object is a Port, and this special case is one of the reasons to introduce Ports.

4.2.2 Basic Components

Blackbox views of basic components form the basic building blocks of the hiearchical composition of components. The structure of a Basic Component consists of a set of classes and their relations (as defined by the OMEGA kernel model language), a subset of some of its classes associated with its Ports, a set of (Provided and Required) Component Interfaces which are associated with its Ports (same as with Blackbox Components), and, finally, connections between Provided and Required Component Interfaces. There are no other components inside a Basic Component. Some of the classes inside a Basic Component have nothing to do with Ports, some of the classes are helper classes that help realize Ports, other classes have (part of) their interface(s), expressed in Component Interfaces, exposed to the outside world via Ports. That ouside world consists of other components, as discussed in Sect. 4.2.4.

Figure 4.1 shows the UML meta–model of the representation of the internal structure of a Basic Component, together with that of the Blackbox Component we saw earlier. Just like a class definition in OO has a class name, we define a component model name for a component model. A model for a Basic Component and a model for a Blackbox Component refer to the *same* component model if their component models have the same name. A Blackbox Component model and a Basic Component model with the same name will have the same Provided Component Interfaces and the same Required Component Interfaces.

4.2.3 Extensions to the OMEGA UML subset discussed so far (Fig. 4.1)

Provided Component Interface

A Provided Component Interface is modeled as a specialization of a UML interface. The Provided Component Interface can be realized by a Class via a Port, or by one of the Ports of a Blackbox Component (thus hiding, encapsulating, classes in a Basic Component).

Required Component Interface

A Required Component Interface is a specialization of a UML interface. The Required Component Interface can be required by a Class via a Port, or by one of the Ports of a Blackbox Component.

Port

A Port is a specialization of a both a UML Class and a UML Interface. A Port can be regarded as a UML Class, whereby the interface of the Class is known but the name of the Class is unknown. Creating an instance of a Port means creating an object with a known interface, but without the need of knowing the class of the object.

One Port can group several Component Interfaces, both Required and Provided. More than one class inside a Basic Component can be involved realizing a Port. More than one class can require services from outside the component via a Port. In Basic Components there can be dependency relations and realization relations from classes to Ports, as shown in Fig. 4.1 with the *required* and *provided* rolenames respectively. It is possible that one or more classes realize a port and one or more other classes depend on the same port. A Blackbox Component, and so by inheritance also a Basic Component, can have several Ports. A designer can give names to Ports so they can be identified when the same Port is appearing in different diagrams.

Blackbox Component

A Blackbox Component is a model for a component where only its Ports and the Provided and Required Component Interfaces are visible from the outside. It is a specialization of a UML Classifier.

Basic Component

A Basic Component is a component consisting of classes and their relations as defined in the `OMEGA` kernel model [OME]. Some of the classes are associated with Ports: they can depend on them or they can realize them. The Basic Component is a specialization of a Blackbox Component.

> A Basic Component inherits Ports with their Provided Component Interfaces and Required Component Interfaces from a Blackbox Component.

We would like to give a few extra remarks about the associations in Fig. 4.1.

A class can depend on several Ports. Since Ports inherit from UML interfaces we can draw dependency relations from classes to Ports. A class depending on a Port implies that that Port depends on a Required Component Interface. Via a dependency relation from a class to a Port a class exports information about its implementation in terms of required services. Our Component Interfaces inherit from UML interfaces so, when drawing a component diagram, we can use the UML dependency relation from Ports to Required Component Interfaces outside of the component, or the corresponding lollypop notation. These notations are the same as for the Blackbox Component. For every Required Component Interface there will be one Port depending on it. For every Port there can be several classes depending on it. In the case of Basic Components the same special dependency relation from Provided Interface to Required Interface on the same component is possible like mentioned in the case of Blackbox Components.

A class can realize a Port by itself, or it can realize "part" of the Port: there can be more than one class realizing the same Port. A class can also be involved in the realization of several Ports. In designing component based systems this is where the designer can abstract from the underlying class diagram; future versions of the component design can use a class diagram that is different from earlier versions, corresponding to a new version of the implementation of the component. In the diagram we can draw realization relations from classes to Ports. A class realizing a Port implies that that Port realizes a Provided Component Interface, drawn with a UML realization relation from a Port to a Provided Component Interface that is outside the component, or with the lollypop notation. For every Provided Component Interface there will be one Port realizing it.

Next we describe some further aspects of the classifiers in our component model.

The outside of a Basic Component is drawn like a Blackbox Component, the inside of a Basic Component uses UML 1.4 syntax for class diagrams, with dependency relations and realization relations from classes to Ports. Figure 4.3 shows an example Basic Component. It models a Client component that needs services from a Server component via the IServer interface. The Client offers services to outside components like `receivePercept` which is used to send data to a Client. The SWC class inside the Client provides the clients' services in this specific application. The XMLRPC class inside the Client is for making a connection with a Server component via its IServer interface: it provides the protocol used between components and it establishes proxies when they are needed.

Both classes and components can engage in provided–required relationships, since a class can be a Port. Here we call the interfaces between them *Component Interfaces* to make clear we are talking about components and to ensure the interfaces adhere to the rules we give for Component Interfaces in this section. There can be classes *and* components depending on the same Required Component Interface via the same Port.

A Required Component Interface can not depend on something inside a Basic Component. That would be a design error since the component can supply the needed services by itself.

An interface (an ordinary UML interface, not a Component Interface) in the class diagram inside the Basic Component that depends on something from outside, should be modeled as a Component Interface. The designer is free to allow class libraries from outside that can be used inside a Basic Component, but this would be a strange design: it would raise the question why the designer did not turn the interface into a Component Interface. Although it would be a design some would frown upon, we do not want to go as far as to forbid it completely. There can be practical considerations, for example it could be difficult to use an existing library in a component framework setting because there is not enough library source code available.

4.2.4 Component Systems

Now that we have defined Basic Components and Blackbox Components, we can finally define components that have other components inside: a Component System can be viewed as one component but with an internal structure consisting of Blackbox Components. This recursive definition gives us the hierarchical structure we need for modeling component based applications.

We use diagrams for Component Systems to show how components are used together, and to show what components need services from which other components. In Component System diagrams only components, their Ports, and their Component Interfaces and their connections are shown, using the

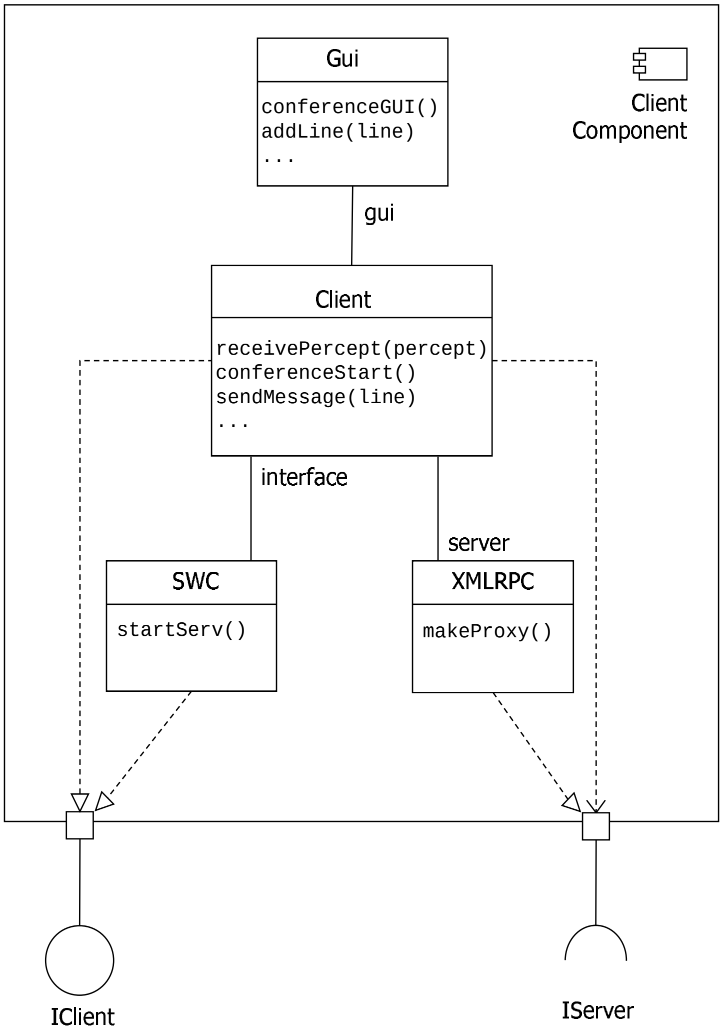


Figure 4.3: A Basic Component, with a class diagram inside

notation of Blackbox Components extended with notation to connect Blackbox Components. Such connections are given by dependency relations from a Required Component Interface to a Provided Component Interface, necessarily crossing a border between two components. There should not be a dependency relation from a Required Component Interface to a Provided Component Interface on the same component. This would mean that the Required Component Interface is depending on services from other components while the component can provide for these services by itself, so the Required Component Interface is redundant. A dependency relation of a Provided Component Interface to a Required Component Interface on another component can not readily be given a useful meaning: we consider such a relation a syntax error.

A Component System is a specialization of a UML Classifier. As described in the section about Blackbox Components, a future UML version could well give us a more suitable entity to specialize from. A Component System is also a collection where its internal Blackbox Components can be seen inside. This is shown in Fig. 4.1 with the generalization association to UML Classifier and the composite association from Component System to Blackbox Component. A Component System also has Ports via the Blackbox Component inside. This means that some of the Ports of its Blackbox Components are exported in order to serve as its interaction points. In fact, a Component System also has a *blackbox view*: the Component System as a whole can be seen as a Blackbox Component that has the same *name* as the Component System and the same Provided and Required Component Interfaces and the same Port names, but nothing can be seen inside. Blackbox views of Component Systems provide levels of abstraction: a Component System can contain Blackbox Components that are Component Systems themselves.

Figure 4.4 shows an example Component System. It models how the Client component from Fig. 4.3 is connected to a Server component. The designer has also decided to turn the Graphical (GUI) User Interface part of the Client (which was just a class called `GUI` in 4.3) into a separate component, so the GUI of the client can be changed and replaced easily. The GUI component forms a Component System together with the Client component that could also be viewed as one "GUIClient" Blackbox component. Also, all the components in Fig. 4.4 together form a Component System.

Inside a Component System, a Provided Component Interface of one component can provide for several Required Component Interfaces of other components, and a Required Component Interface can depend on several

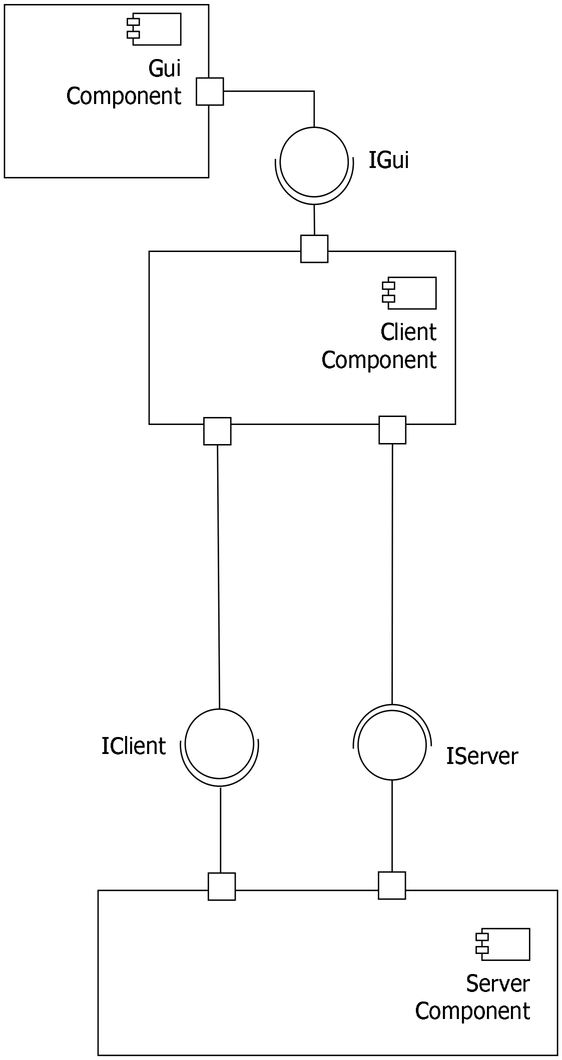


Figure 4.4: A Component System

Provided Component Interfaces.

Figure 4.1 shows the meta–models for Basic Components, for Blackbox Components, and for Component Systems in one figure. They have been combined into one figure so the reader is given a good general overview.

What is not specified in the meta–model however, is the following important condition which ensures encapsulation: The signatures of operations and signals in a Component Interface should only contain standard classes and data–types from the implementation language (for example the OMEGA kernel model language) and classes that are exported as Ports. Note that thus encapsulation is ensured because we do not allow inheritance relationship across component borders (we only allow dependency relation between Required and Provided Interfaces).

4.3 Runtime Behaviour

In OMEGA we associate with each class a statechart which describes the runtime behavior of its instances. Because ultimately an OMEGA component model can be flattened to its underlying class diagram (this reduction is formally worked out in an OMEGA deliverable), this association completely defines the runtime behaviour of a component. It is important to observe that here we are referring to the runtime behaviour which abstracts from the actual deployment on a specific execution platform.

The labels on the arrows in these statecharts contain OMEGA action language and they are of the form `[guard] trigger / action` where guard is a boolean expression, trigger is an event or a method name with its parameters and action is a primitive action in the OMEGA action language. These primitive actions use standard OO dereferencing with the dot notation and are of the form `a := a0.a1`, `a0.a1 := a`, `return := a` and other simple statements; see the OMEGA kernel model document [OME] and [DJPV03] for a complete enumeration.

In our model the required services of a component are specified by means of interfaces, as described formally in the meta–model. Acquiring an object that provides the functionality of a component with interface `I`, requires the instantiation of a class whose interface is known but not its definition (which is given in another component). Therefore, in OMEGA we have extended the UML action language used in statecharts with this notion of "instantiable interfaces", that is, in the action language we allow assignments `x :=`

`new(I)`, where I is a (required) interface. This way we can make instances of classes that are defined in other components, but without the need to know the name of the class in the other component (which would be impossible in the case of a future implementation of the other component). There are several ways to actually implement this scheme, in Fig. 4.3 it is the SWC class that makes sure that the correct class is instantiated in a Client component. This class is simply called Client in the figure, but in a future version it could be a class called NewClient.

As such we are instantiating a class but we only know the interface (`I`) of the class, we do not know the name of the class nor its implementation. However, in the case of one *complete* component application it *is* known which class implements `I` so we can simply compile `x := new(I)` into the corresponding `x := new(C)`, where class `C` implements `I`.

Our component model thus abstracts from the underlying component framework (for instance CORBA). To provide services of class `X` to other components in a component framework, the designer can assume that a class `Y` exists that does introduce the services of `X` to the component framework. This class `Y` is a class that realizes a Port in the model. There are several ways class `Y` can do this: it can accept an object that is an instantiation of class `X` as a parameter to one of its methods and delegate the desired services to this object; or it can use a "mixin" technique that extends the interface of class `Y` with the desired services of class `X` and instantiate a new object of type `XY`; or it can create a new object of class `X` and delegate desired service calls to this object; or it can use another mechanism.

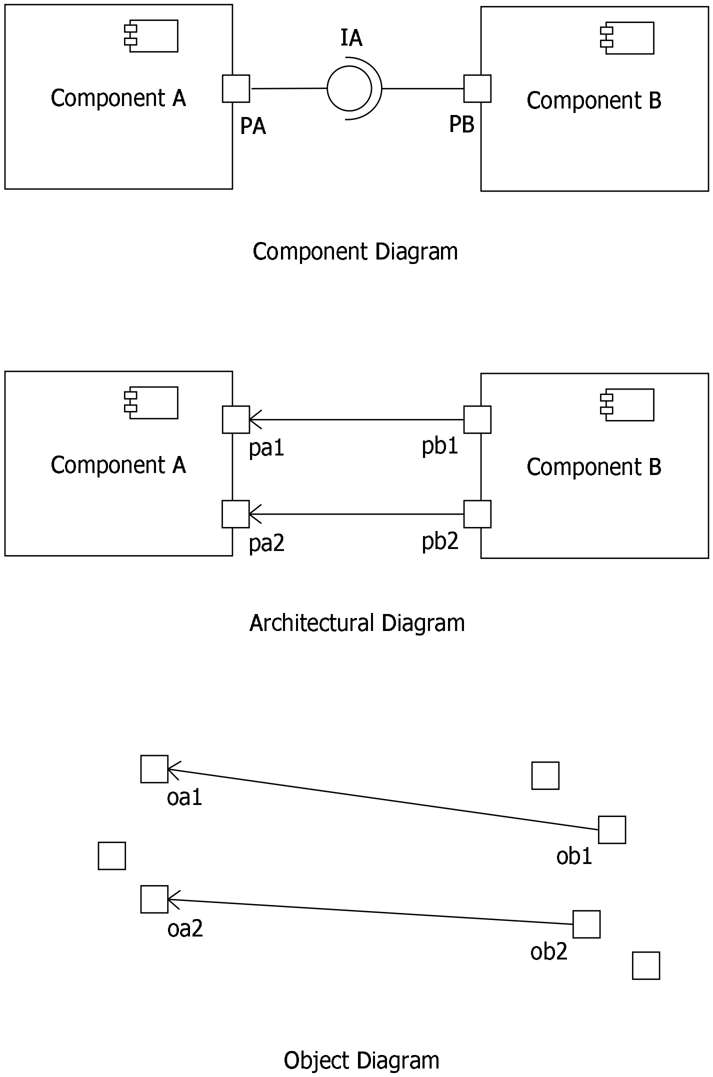
To summarize, we do not have an explicit notion of "instances of components" but we only have instances of Ports. Of course it is possible to design software in such a way that objects, referenced by variables in the source code, are created that act like instances of components. But we do not enforce creation of component instances: if the designer wants to model a component as a software abstraction only, it is possible.

4.4 Architectural Models

Architectural diagrams show component interaction configurations. They are snapshots that can be used to describe the initialization of a component system, invariant properties of the configuration, and others useful runtime characteristics. In OMEGA we will use a very restricted subset of OCL for expressing the semantics of architectural diagrams formally. In architectural diagrams components interact by means of Ports. These Ports are different from the Ports in a component model, the emphasis here is on *objects* with a specific interface, not *classes*. If there is a possible confusion then we will call the Ports in architectural diagrams Runtime Ports, and Ports in component diagrams Component Ports. In an architectural model the Runtime Ports can be viewed as named interfaces. Although normal OO interfaces are not instantiable, in our model, as discussed previously, Component Interfaces are instantiable via their Component Ports, resulting in Runtime Port objects: this gives us the possibility to model component interaction like other OO interactions. In an architectural model, Runtime Ports are instantiated interfaces. In an actual implementation a Runtime Port can be an object that delegates to several other objects, or it can be a channel–like object with an address and location; what choice is made exactly is not important for the design: it is an object that realizes a Component Interface. In our model we define interaction between Ports as standard OO interaction.

Figure 4.5 shows an example architectural model, together with a component model above and an object diagram below. In the component model can be seen that `Component B` requires services of `Component A`. In the architectural model can be seen that there are, at some point in runtime, exactly two Ports of `Component B` connected with `Component A` and they are using the same services but from different Ports. The connections are *directed* from requiring to providing Port. In the case of two Ports that use services from each other an undirected connection can be shown by drawing a line without an arrow. The Ports of `Component B` are instances of `PB`, the Ports of `Component A` are instances of `PA`. Such a configuration can be specified with OCL, but the architectural model is also useful: it is easier to draw a picture like this than to have to learn OCL. The bottom diagram in Fig. 4.5 shows an object diagram that corresponds with the architectural model above. It shows the objects that realize the Runtime Ports. This makes it clear that the components in an architectural model are not software abstractions but collections of objects.

Figure 4.6 shows the meta–model for architectural models. For the connections between Ports we use UML associations, we are awaiting the next MOF to decide what the final meta–model will look like, viz. the `Object Collection` entity.



Component Diagram

Architectural Diagram

Object Diagram

Figure 4.5: The same application modeled with a Component Diagram, an Architectural Diagram and an Object Diagram

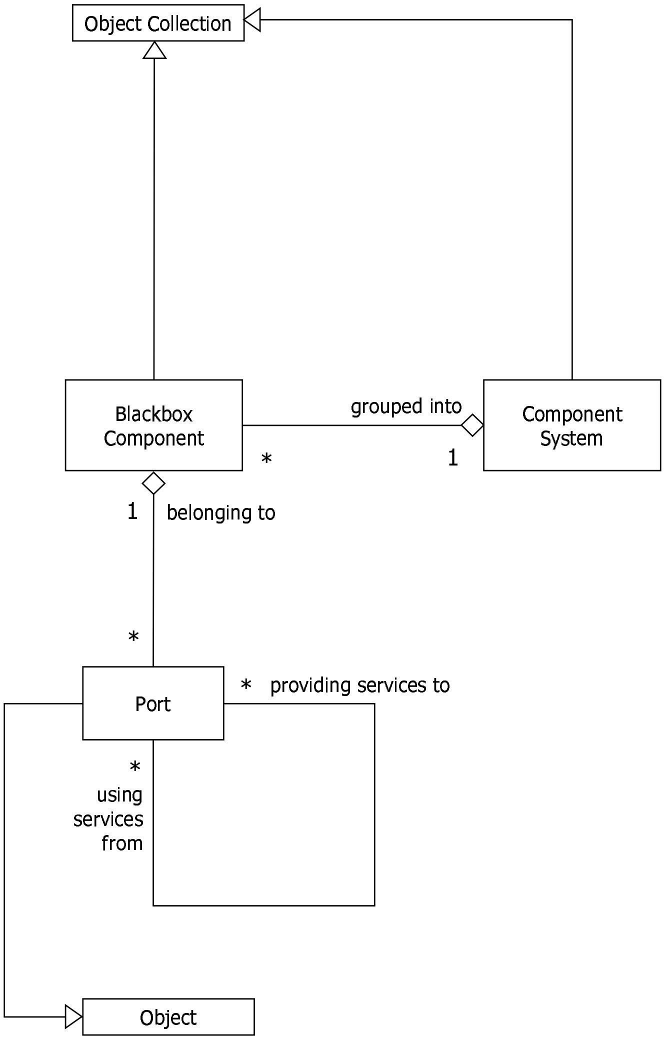


Figure 4.6: The Meta–Model for architectural models

4.4.1 Overview

As mentioned in the Introduction, in the OMEGA project we have chosen a subset of UML 1.4 so we will be able to use formal techniques. This subset consists of class diagrams and object diagrams for structural modeling, and statecharts and OCL for behavioral modeling. We also use Live (LSC) Sequence Charts [DH01] in OMEGA but they are not part of UML. The subject of this paper is about the extension of this OMEGA UML subset with components. An overview of all the UML diagrams we now have is available in Table 4.1.

Definition	Deployment	Behaviour
Component Diagrams	Architectural Diagrams	Specification: Statecharts, OCL
Class Diagrams	Object Diagrams	Implementation: Statecharts

Table 4.1: UML Diagrams in the OMEGA project

In our component model we define component diagrams that relate to architectural diagrams like class diagrams relate to object diagrams. When designing OO software, both class diagrams and object diagrams are useful; they serve different purposes: with class diagrams the designer gives a definition view, with object diagrams a runtime view is given. Likewise, both component diagrams and architectural diagrams are useful.

Summarized, *class diagrams and component diagrams are used for modeling definitions; object diagrams and architectural diagrams are used for modeling configurations.*

If a class diagram is complete and if there are constraints added with a powerful enough constraint language like for example OCL, then all corresponding possible object diagrams can in theory be derived from it. A design goals of our component model was to offer similar diagrams and possibilities for modeling components.

To model behaviour in UML statecharts can be used. They are associated with class diagrams and they give an implementation. On the *component level* we can also use statecharts to model behaviour. To be able to model behaviour on the component level requires a different action language than that used in statecharts associated with classes: statecharts associated with components describe the interaction between, and coordination of, different objects, whereas statecharts associated with classes describe the behaviour of one object. On the component level we use statecharts together with OCL to specify behaviour: this is indicated in Table 4.1 in the right–most column, with statecharts in the top row for *Specification*, and statecharts in the bottom row for *Implementation*.

Table 4.1 gives a classification of the UML diagrams we use for modeling components and their relation to the existing class diagrams, object diagrams and statecharts. Together with the explanations in this paper we hope this sufficiently answers often heard questions from users like "When to use what UML diagram?"

4.5 Trace Semantics

In order to provide a semantic basis for the compositional verification of components, in this section we briefly outline the formal trace semantics of components which describes the external observable behavior of a component as determined by its ports. OMEGA deliverable D1.1.2 describes a formal operational semantics of UML class-diagrams where the behavior of the object instances of each class is described by a statechart ([Har87]). This semantics abstracts from the actual deployment unto a specific execution platform. It is formalized in terms of a translation relation on object-diagrams which specify for each existing object the values of its attributes and the values of some system variables which encode some relevant control information (such as the current state in the associated statechart).

On the basis of this operational semantics for UML class diagrams we can define inductively the trace semantics of a component. For basic components, the internal structure of which is given by an UML class diagram, we extend the above transition relation to a labelled transition relation

$$\sigma \xrightarrow{\lambda} \sigma',$$

where σ and σ' denote object-diagrams which represent the internal object-structure of the component before and after the transition and λ is a label indicating an internal computation step or an external event. An internal computation step is indicated by τ. An external event is of the form

$$o.m(o', p_1, \ldots, p_n),$$

where

- o denotes the callee of the event,
- o' denotes the caller,
- $p_1, \ldots, p_n$ denote the actual parameters, and finally,
- m denotes the kind of message.

Note that adding the caller as an explicit parameter, together with the encapsulation condition, implies that all interaction between components is via their Port instances. That is, we do not allow an internal object of a component (i.e., objects that are not instances of a Port class) to call the provided services of a Port instance of another component.

For technical convenience only, we restrict in the current presentation to messages of the following kind:

- *op*, which indicates the invocation of an operation call,
- *return.op* which indicates the return of an operation call, i.e., $o.return.op(o', v)$ denotes the return of a call from o' to o with return value v.

Object-instances are denoted by pairs of the form (id, I), where id denotes the identity of the object and I denotes its interface. An object is external (to a given component) if its implementation is not known, that is, if its interface is a Required Component Interface. For an internal object we identify its interface with its class using existing UML. In an external event either the caller of callee denotes an external object.

The above transition relation generates the traces of external events of a basic component. The global behavior of a system of components $Comp_1, \ldots, Comp_n$ we can now describe compositionally in terms of the traces of external events of its components by means of a projection operator: Given a global trace of events t, a component *Comp*, the trace $Comp(t)$ denotes the subtrace of t consising of external events involving port-instances of *Comp*. More specifically, we have also to rename the identity of an external object (id, C) to (id, I), where I is the Required Component Interface of *Comp* provided by port-class C (defined by another component). It is important to observe here that at the level of a component system the (high-level) dependency–realization relations between Component Interfaces provide information about which are possible events. Namely, an event like $o.m(o', ...)$ is possible if $o = (id, C)$ and $o' = (id', C')$ implies that there exists a connection between the ports C and C'.

Definition 1 *Given a system of components* $\mathcal{C} = \{Comp_1, \ldots, Comp_n\}$ *we define*

$$Trace(\mathcal{C}) = \{t \mid Comp_i(t) \in Trace(Comp_i), i = 1, \ldots, n\}.$$

Note that $Trace(\mathcal{C})$ specifies the global behavior of the component system $\mathcal{C}$. We can define the externally observable behavior of the blackbox view of $\mathcal{C}$ in terms of a hiding operator which removes all internal events.

The above trace semantics forms the basis of a corresponding trace logics for specifying invariant properties of the traces of components (see also OMEGA deliverable D1.2.1). We are working on a tool based on the semantic tableaux method which allows to check the compatibility of the trace-invariants of the components in a system in terms of a logical formulation of the above compositional definition.

4.6 Modeling with Components

In this section we discuss more practical aspects of modelling applications with components in OMEGA. In the OMEGA User Guide the concrete syntax for component models can be found, here we suffice to say that in the absence of a CASE tool that supports UML 2.0 components, the correspondence between component diagrams and class diagrams gives the possibility to use class diagrams to model components. It will give the user a little more administration to do to remember which diagrams are for components and which are for classes. Likewise, object diagrams can be used to model architectural diagrams.

The notion of an interface as specifying a set of provided and required operations (or signals), respectively, supports a development process of component-based software systems in UML that distinguishes two main levels of abstraction, promoting a separation of concerns between the external communication of data and the internal processing of data. At the higher level of abstraction, a system is described in terms of the interactions among its components, abstracting from their actual internal implementation. This level provides the black-box view of a component. The lower level concerns the modeling of the data-processing aspects within each component. The resulting *hierarchy* object-class-component provides a natural and powerful scheme for distribution and abstraction, hiding and structuring the complexity of large distributed object-oriented software systems. More specifically, the dynamic creation of any number of port-instances allows a component to interact in a really distributed manner. This is to be contrasted with the run-time notion of a component as a group of objects associated with an instance of an active class which share a single thread of control and an event queue of asynchronous signals.

The additional structuring and abstraction mechanism provided by the notion of component allows a considerable simplification of an underlying kernel model language like in OMEGA, which basically consists of removing the distinction between active and passive classes. More specifically, every instance of any class has its own single thread of control and its own event queue. Acceptance of signals and operation calls by an object are defined only in terms of the local state of the object itself. Objects are grouped together only by means of the static structuring mechanism of components.

This simplification of the OMEGA kernel model language (and its semantics) allows both for more transparant models and more efficient verification techniques. The additional complexity provided by components then can be dealt with by means of the application of compositional verification techniques.

4.6.1 Examples of software developed with the component model

We have developed several applications with our component model to see whether it is useful practice. New versions of the component model reflected the experiences with the designs. Most figures in this paper are based on an example called "Conference". With this system users can have a distributed conference where they communicate with each other by typing messages, somewhat like IRC chat on the internet. The system consist of a server application and a client application. The central server of a conference can be setup by any of the users and is accessible at a HTTP URI via XMLRPC. This means that the client application that is used to connect to the server can be written in any programming language. We have example clients written in Python and in Java. As another example the OMEGA partner FTRD has modeled their OMEGA application with our component model.

Using the component model turned out to be a natural and intuitive way of designing software. The software engineer can concentrate on high level system designs first and design lower levels later. Of course this could also be done with a class hierarchy but there the designer has for example no "instantiable interfaces" (our component ports), forcing the designer to make decisions about class names and class hierarchies and the like much earlier in the design phase.

The example applications are available at our OMEGA component model website [Jacc].

4.7 Conclusion and related work

In this paper we have presented a model for components to address architecture and component based development. The main idea is that a component is an abstraction, like a class or a module, A component is a grouping of classes, some internal and others, the so–called Ports, denoting interaction points with the component environment. Only Ports are visible to the environment. Each Port is attached to a set of provided and required interfaces.

Components are used in two type of diagrams: *component diagrams* and *architectural diagrams*. Component diagrams are for describing the structural dependencies among the provided and required interfaces of components in a system, while architectural diagrams are for the description of the runtime architecture of the system. In architectural diagrams Port instances are linked together by means of UML associations which indicate that the connected Port instances know each other.

Considering component as an abstraction of its internal parts, in contrast to the concept of component used for deployment in UML 1.4 [SP99], implies that components are not units of instantiation and do not need to have a unique run-time identity. Moreover, having Ports as instantiable interfaces, in comparison with the recent component model proposed by the U2 partners for UML 2.0 [U20], has the advantage of permitting the existence at run time of multiple Ports with the same set of interfaces per component, each Port attached to the necessary number of runtime links. These runtime links are modeled as connectors in UML 2.0.

Our model offers a coherent view for the design of architecture and component-based systems. Components serve as a naming mechanisms for abstracting from the internal parts, interfaces as declaration mechanisms of services (either provided or required) and Ports together with the dependency–realization relations as abstraction mechanisms of object interactions.

Architecture description languages (ADLs) define also high-level concepts for the design and modelling of architectures of systems, such as components, Ports, and configurations. A large number of ADLs have been proposed, some of them with a sound formal foundation. We only mention here Wright [AG97], Rapide [LKA$^+$95] and ACME [GMW97]. Closer to UML are the architectural descriptions provided by SDL [BH89], ROOM [SGW94] and UML-RT [Sel98] (the latter is in fact a UML profile interpreting ROOM concepts in terms of UML stereotypes). In [GCK02] and [MRRR02], several strategies for modelling components and other architectural concepts within UML are investigated, with as conclusion that these concepts are hard to describe in UML as it is.

Many models for components have been proposed in the last years, some informal and remaining within the realm of the existing UML (see for example [CD00]), and others founded on a logical and mathematical basis (e.g. Broy's component model based on streams of messages [BS01]. Similar to Broy's component model, the semantics of our model is also based on sequences of messages (like those used for the semantics of CSP [Hoa85]). However OMEGA components have dynamic aspects (e.g. Port instances) not fully covered by Broy's model. Moreover our component model is a conservative extension of an object-oriented model and therefore it requires the addition of only a couple of extra concepts to the standard UML 1.4. It is interesting to note that these additional concepts are also required by the component model proposed for UML 2.0 by the U2 partners [U20]. As described above, however, the semantics of these concepts is different between the two models.

Other interesting approaches are the one taken by Catalysis [DW98] and the precise UML group [pUM]. In OMEGA we are currently investigating the relationships between these approaches and our model and possible ways of integration.

Finally, we have shown how to exploit in a formal mathematical manner the hierarchical structure of components in compositional verification. Currently, we are working on the development of a tool for checking mutual consistency of the behavioral specifications of a set of components.

Acknowledgement The author is grateful for the input and helpful discussions with members of OMEGA, especially Frank de Boer and Marcello Bonsangue, in the design and evaluation of this component model.

Chapter 5

Component Coordination in UML

Authors: Frank de Boer, Marcello Bonsangue, Joost Jacob

5.1 Introduction

Modeling is an essential part of large software projects. The Unified Modeling Language (UML) has become the de-facto standard language for specifying, modeling and documenting software systems, visualizing software systems. The basic innovative ideas of UML, which are the main reasons for its popularity, are the unification of the concepts and notations used in the life-cycle of software development as well as the recognition of the importance of modeling and analysis as a means to improve quality. UML consists of a number of diagrams used for expressing the goals of the system (use case diagrams), for specifying the structure of the system (class diagrams) and the behavior of the system (state diagrams, activity diagrams, sequence diagrams).

In this paper we introduce a formal model of components in UML. This model has been developed in the context of the European IST project OMEGA. The aim of this project is the correct development of real-time embedded systems based on formal techniques. The approach followed is based on a formal semantics of a suitable subset of UML which includes class and state diagrams, a version of the Object Constraint Language, use case diagrams, and live sequence charts (an extension of UML's sequence diagrams [DH01]). The semantics of the UML subset, here called Ω-UML, is defined in terms of a formal interleaving semantics obtained by associating with each model of Ω-UML a symbolic transition system [DJPV03].

Our component model generalizes the basic concepts of object-orientation by providing additional structuring and abstraction mechanisms which allow a modeling discipline and the application of formal techniques based on interfaces. More specifically, it allows to structure the class diagrams of a UML model into components and to abstract from the internal details of these encapsulated class diagrams. Because of the encapsulation provided by components we can compose them hierarchically in a natural manner.

In this paper we also discuss the formal semantics of our component model. First we discuss the formal relation between a system of components and the underlying UML class diagrams. This relation is defined in terms of a reduction which 'compiles away' the additional structuring and abstraction mechanisms provided by components. However, we also show how we can describe the externally observable behavior of a component at a high-level of abstraction and compositionally in terms of its structuring and abstraction mechanisms. This latter view provides the formal justification of the modeling to interfaces discipline and it provides a formal basis for the application of formal techniques to components.

Furthermore, we discuss different coordination patterns in the context of our component model. First we show how high-level components can be used to model the low-level coordination patterns underlying the computational model of Ω-UML. These coordination patterns form an intricate combination of the asynchronous communication supported by an event-driven computational model (along the lines of the Actor model [Agh86]) and the synchronous communication supported by the usual rendez-vous mechanisms of operation calls in object-orientation. Finally, we show how to generalize our component model to a model of component coordination based on mobile channels which allow a clear separation of concerns between coordination and computation.

This paper is structured as follows: Section 2 introduces the component model. In Section 3 we discuss the formal semantics of our component model. Section 4 then proceeds with a discussion of how to model the low-level coordination patterns underlying Ω-UML by means of high-level inter-component coordination. Finally, Section 5 discusses a generalization to mobile channels. In Section 6 we draw some conclusions.

5.2 A component model

In this section, we introduce an extension of UML addressing the area of component-based software systems. Following Szypersky [Szy02], we see a software component as a *unit of composition* with well-defined interfaces, that can be independently developed and subject to composition by third parties. In the context of UML, this means that we consider a component as a mean to provide a high-level software abstraction like that of a *module*, which encapsulates its internal structure and which provides interfaces specifying the provided and required operations. The rationale is to provide a structuring and abstraction mechanism which allows a modeling discipline based on interfaces.

More technically, a *component* is a UML classifier, which is intended to be self-contained and re-usable during development and deployment. It is identifiable by a name but it has no attributes and operations. It cannot be instantiated or be part of associations, but it can be generalized since it has a type, defined by the set of its provided and required interfaces. This means that a component is a *unit of substitution* that can be replaced by a component that offers at least the same provided interfaces and demand at most the same required interface.

A *component interface* is just a UML interface, that is, a non-instantiable classifier with operations and attributes. We distinguish between two kind of component interfaces: *required interfaces* and *provided interfaces*. A provided interface specifies a set of operations that the component offers to the environment. A required interface specifies a set of operations that are needed by the component to guarantee the correct functionalities of some provided interfaces. We allow for generalization relations among component interfaces.

A component is also a package, and therefore a structural em unit of abstraction of the classes realizing its behavior. Other UML elements may be owned by a component. In particular, other components may be owned by a component allowing for hierarchical specifications. Encapsulation of the internal structure is guaranteed because interaction points with its environment are exclusively defined via ports, the software concept equivalent of the hardware port on a board.

A *port* is a class and also an interface, that is, it is an instantiable interface. A components owns a set of ports, and each port owns a set of the component provided interfaces, and a set of the required interfaces. The same component interface may be owned by more than one port. The set of ports defines the border between the internal implementation of the component and its environment. Internal classes may realize a port or depend on a port.

Incoming communications defined in the provided interface of a port are handled within instances of an internal class of the component realizing that port. If a class realizes a port it realizes also one of its provided interfaces. We assume that at most one internal class may realize a provided interface of a port (but we allow for different classes to realize the same provided interface if each class realize a different port). A port introduces an indirection, and each request of instantiation for that port is resolved at run-time by instantiating an internal class realizing a provided interface owned by the port (which class is resolved statically by the type of the object expected by the requester from the instantiation). This indirection mechanism abstract from the actual implementation of an operation and allows for a very late binding of an operation implementation with its declaration in a component interface. We call *port instances* the object instance of internal classes instantiated by a port.

If an internal class depends on a port then it depends also on one of its required interfaces. From the environment point of view, outgoing communications of an object instance of an internal class are identified with communications from the port owning the required interface on which that class depends. Communications at the border (that is, between two port instances) are observable.

A component has two structural vies: a black-box view and a white-box view. In the *black-box view*, only the component provided and required interfaces and their grouping into ports is visible. Optionally, behavioral elements such as a state machines may be attached to each port, to define more explicitly a sequence of operation calls. For a black-box component it must hold that every type or class used in a provided interface must be declared in one of the provided or required interface of the component itself. This self-containment property, together with not allowing generalization across the border, ensures a complete encapsulation of the internal implementation. Notationally, a black-box component is drawn as a classifier rectangle with in the right hand corner a component icon: a rectangle with two smaller rectangles protruding from its left hand side. Ports are shown as small squares on the edge of the component rectangle, with association to interfaces, shown as labeled ball and socket for the provided and required interface interface, respectively. Provided and required interface can also be shown more explic-

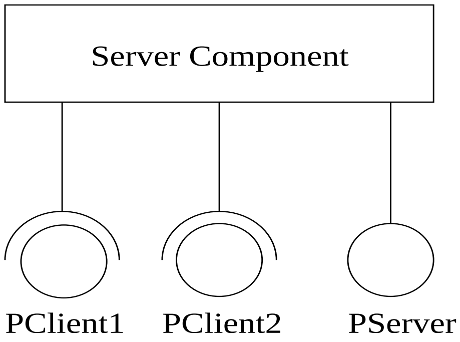


Figure 5.1: A black-box view of a component

itly as the classifier rectangles. Figure 5.1 shows an example of a component with two ports: one with a provided interface and another with a provided and a required interface.

Black-box views of components are used in *component system diagrams* to visualize the structural connections in a component-based system. There can be dependency relations from a provided interface of a component to a required interfaces of another component (but not vice-versa). Optionally, a coarser specification of the structural collaboration can be made using *connectors*. A connector is a specialized association between ports, used to indicate that all required interfaces of a port must be compatible with the provided interfaces of the other connected port. The wiring between components in a system is used at run-time for the instantiation across components. If an internal objects (i.e., an instance of an internal class) requests the instantiation of a port P with interface I on which it depend, then this request is resolved at deployment time in a request for instantiation of the port Q with a provided interface wired with the required interface I of P. In other words, a port through its required interfaces act as placeholders for port names that become known only at deployment time, when connectors or dependency relations are statically fixed in a component system diagram.

Figure 5.2 shows a component system diagram. Dependency relations between provided and required ports implicitly given by the ball-in-socket notation. The association between a port of component B and one of component C is a connector: the set of provided and required ports of those ports must be compatible.

In the *white-box view*, the internal elements of a component are revealed,

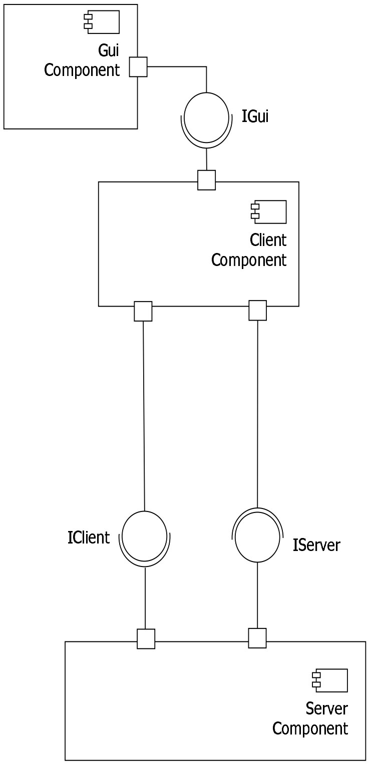


Figure 5.2: A component system diagram

in order to show the implementation of the external behavior of the component ports. To this end, dependency and realization relations must be given between internal classes or internal components and the component ports: a dependency relation provides information about an internal class implementation in terms of its required services. An internal class can *depend* on several ports. Since ports are UML interfaces we can draw dependency relations from an internal class to a port, with the intended meaning that the class depend on one of the required interface of that port. On the other hand, a required component interface or a port cannot depend on something inside a component. That would be a design error since the component can supply the needed services by itself. Since ports are UML interfaces, we can draw realization relations from internal classes to ports. An internal class can *realize* several port, but port can only realize provided interfaces. A connector between the port of an internal component and an external port is used to graphically show the export of a port of an internal component to the environment.

Figure 5.3 show the white-box view of component A: the upper port is connected with a port of the internal components B, while the other port is realized by class C. Class C depends on the same port and also on a provided interface of the internal component B.

Run-time component interaction configurations are modeled in architectural diagrams. They are snapshots that can be used to describe the initialization of a component system, invariant properties of the configuration, and others useful runtime characteristics. In architectural diagrams only instances of ports, ports and components are shown, together with their relationships. Instances of ports handle all interactions from the environment into the component they belong to, as well as the interaction between different port instances. Interaction from the inside of a component to the environment is handled by the ports of the component (and not by their instances).

In Figure 5.4 we show an architectural diagram of the component system depicted in Figure 5.2. Port instances are represented by filled squares, while port classes are denoted by plain squares. Arrows denote directed relations either between port instances (representing the possibility of executing operation calls from a port instance to another one), or from port classes to port instances (representing the possibility of executing operation calls from the internal of the component to the port instance of another component).

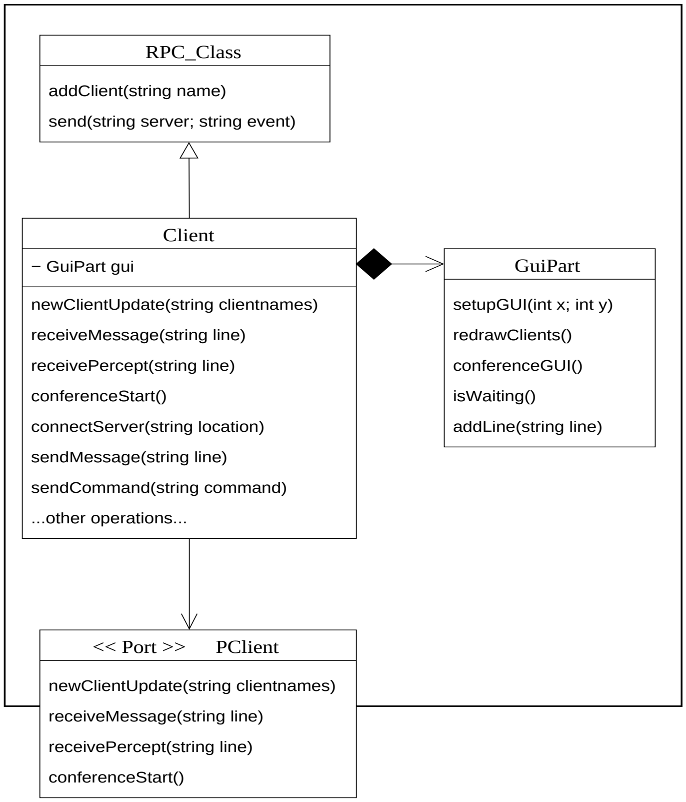

RPC_Class
addClient(string name)
send(string server; string event)
Client
− GuiPart gui
newClientUpdate(string clientnames)
receiveMessage(string line)
receivePercept(string line)
conferenceStart()
connectServer(string location)
sendMessage(string line)
sendCommand(string command)
...other operations...
GuiPart
setupGUI(int x; int y)
redrawClients()
conferenceGUI()
isWaiting()
addLine(string line)
<< Port >> PClient
newClientUpdate(string clientnames)
receiveMessage(string line)
receivePercept(string line)
conferenceStart()

Figure 5.3: A white-box view of a component

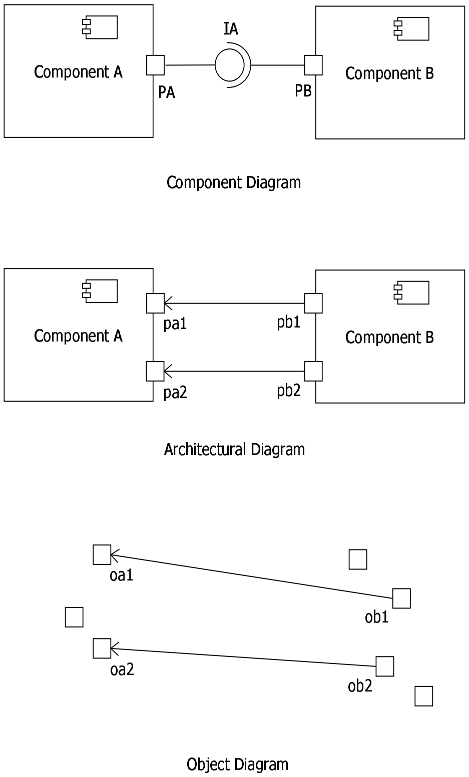


Figure 5.4: An architectural diagram of a component system

5.3 Ω-UML

Building embedded real-time systems of guaranteed quality, in a cost-effective manner, is an important technological challenge. There is a general agreement that a means to achieve this is a model-based approach. UML aims at providing an integrated modeling framework encompassing structural descriptions, as well as behavioral descriptions. Although there is a large number tools available that implement a dynamic semantics of UML, none of these tools integrates state-of-the-art formal validation tools, as required in many industrial sectors for a proper development process.

The aim of the European IST project OMEGA is the correct development of real-time embedded systems based on formal techniques. The approach followed is based on a formal semantics of a suitable subset of UML which includes class and state diagrams, Object Constraint Language, use case diagrams, and live sequence charts (an extension of UML's sequence diagrams [DH01]). The semantics of the UML subset, here called Ω-UML, is a formal interleaving semantic obtained by associating with each model of Ω-UML a symbolic transition system [DJPV03]. Due to space restrictions, we only sketches here the main concepts of Ω-UML and their intuitive semantics.

Similarly to the standard UML [SWB03], classes in Ω-UML may be either active or passive. An *active object* (i.e., an instance of an active class) is like an event-driven task, which processes its incoming requests in a first-in-first-out fashion. It has only one thread of control, so that active objects are internally sequential. As only one message may be treated at a time, there must be a mechanism for queuing the calls. For a *passive object* only one operation may be active at a time, and, although passive, it has certain degree of control over the invocations made toward them, as explained later in this section. Furthermore, in Ω-UML all objects are assumed to be *reactive*, that is, their behavior can be made dependent on the current state of the system.

In Ω-UML *state-machines* are used to describe the computational behavior of the instances of a class. These state machines are composed of transitions which are labeled by a (guarded) *trigger* and an *action*. A trigger specifies the reception of an *operation call*. An action involves assignments to the attributes, class instantiation and operation calls.

The semantics model of Ω-UML captures three different kinds of inter-object communication: share variables (via public attributes), synchronous (via triggered operation calls, i.e. operation calls whose return value depend on the current state of the system) and asynchronous (via signal events).

The execution of a *synchronous* operation call involves a *rendez-vous* between the sender and the receiver of the call: First, the sender and receiver of the call have to synchronize on the execution of an operation call by the sender and a corresponding trigger by the receiver. Such a synchronization results in the sending of the actual parameters which are stored by the receiver in the corresponding formal parameters of the operation. During the execution of the operation by the receiver, the sender is suspended. Upon termination of the call, the return value is send back to the sender, after which both sender and receiver resume their own execution.

On the other hand, an *asynchronous* operation call is stored in the *event-queue* of the receiver. The execution of a trigger involving an asynchronous operation consists of checking whether a corresponding operation call appears as the first element of the event-queue (of the receiver) and storing its actual parameters in the formal parameters. It suspends otherwise.

The above communication mechanisms between active and passive objects is coordinated by means of *activity groups*. Activity groups are run-time components which are created dynamically. Each object belongs to an unique activity group, and each activity group contain exactly one active object at each time.. The objects of an activity group share both one thread of control and the event queue. The sharing of control means that at most one object of the group is executing. Control is passed on by a synchronous operation call to another object belonging to the same group. On the other hand a synchronous call to an operation of an object not belonging to the same activity group suspends the executing object and a fortiori its activity group. An asynchronous operation call to an object will be stored to the event queue of its activity group.

5.3.1 Components in Ω-UML

In order to represent all the aspects important for real-time system design at an appropriate level of abstraction, Ω-UML incorporates the component model introduced in Section 5.2. The model allow the definition of visibility and communication constraints needed for the design of large systems and provide a basis for proper abstraction, compositional refinement and verification.

To properly model the communication mechanism of a component-based system, the action language of Ω-UML is extended to allow actions for port instantiation. They are of the form

$$x := new(P)$$

where P is the name of a port owned by a component and x is an attribute of type compatible with one of the required interface owned by the port P. This action can be executed by any object instance of an internal class depending on the port P.

A component system diagram can be reduced to a large class-based Ω-UML model, using information from the white-box and black-box view of each component involved in the diagram. The basic idea is to recursively (because of the hierarchical structure of a component) transform each component into a the class it encapsulate. The classifiers for ports, component interfaces and components are omitted from this diagram, as well as all relations on which they are involved. Action for port instantiation $x := new(P)$ are transformed into action for class instantiation $x := new(C)$, where C is the name of the class indirectly instantiated by the port to which P is connected.

The basic semantic model obtained in this way is not compositional with respect to the concept of component. It is formalized in terms of a translation relation on object-diagrams which specify for each existing object the values of its attributes and the values of some system variables which encode some relevant control information (such as the current state in the associated statechart). Transitions are labeled by external events or by a label indicating an internal computation step. An external event is of the form

callee.op(caller,parameters)

denoting the call of a (synchronous or asynchronous) operation op by the caller to the callee with a list parameters, or

caller.return-op(callee, actual-parameters)

denoting the return from *callee* of a synchronous operation call *op*, initially called by *caller*. This semantics defines a very fine-grained notion of observability, making roughly every choice point and every interaction between distinct objects observable.

A more abstract notion of observability should be preferred, to capture only those interactions between the component and its environment. This can be obtained by transforming the labels of above transition system as follows. If the callee is a port instance then the transition step should be observable. In case the caller is also a port instance, the label should not change, otherwise *caller* should be renamed to the port on which its class depend. All other steps should be treated as internal steps. By means of a corresponding projection operation on traces of events we can define the traces of a system of components in a compositional manner along the lines of the compositional trace semantics of CSP [SS99] (generalized to components and object-orientation).

5.4 Intra-component coordination

In this section we discuss how our component model can be used in UML to express the coordination mechanism between active and passive classes as prescribed by Ω-UML by means of activity groups. That is, we see a component as a *unit of reaction*, which statically determines the run-time coordination of a reactive system with active and passive objects. Instances of an active classes are like Real-Time Operating System tasks, having their own thread of control, and running concurrently with other instances of active classes. Instances of passive classes are more like sequential objects whose operations are under the control of the active object that controls the caller.

In this section we will concentrate on the basic communication mechanism of UML: objects communicate and synchronize only via synchronous operation calls. For modeling purposes this suffices because asynchronous operation calls can be modeled by synchronous ones.

The basic idea is to replace an activity group from a model of Ω-UML by a component with a port which describes the coordination model of this activity group, that is, port instances will generate and coordinate activity groups. Since activity groups do not provide any encapsulation, the port will own a provided interface for each class specifying the activity group. The port instance is the only object that external objects can interact with, in the style of ordinary message interaction: p.operation(parameters). Further, the port instance p does not have attributes that are accessible by external objects.

Interactions between an external object and an internal object are delegated via the component port instance p: The external object is now interacting with p instead of with an internal object directly: every operation call

callee. op(parameters)

to an internal object callee is transformed into a call

p.op (caller, callee, parameters)

which involves the storage of the corresponding call to the operation op in the pending request table discussed below and which is immediately followed by a transition with a trigger

return-op(return-value)

which will involve the return from the call op by the callee and the reception of the return value.

On the other hand, the internal objects have to be modified so the port-instance can be informed about what triggers they will accept in their current state. We do so by adding to every state of an internal object a loop consisting of a *poll*() trigger and a return statement that returns the names of the triggered operations that can be accepted in the state.

Finally, a component port is an instance of a port class with an attribute which stores operation calls from external objects to objects of the activity group. This pending request table has four columns: every row has entries that consist of the caller object, the callee object, the operation name, and a sequence of actual parameters. Furthermore a port will contain other attributes for expressing certain relevant information about the state of the activity group, e.g., its objects, the executing object, etc.. This additional information will be used in the method *select* that selects an object from the set of internal objects for execution. A port has also a method *choose* that can select an entry in the pending request table. These two methods together will implement the coordination mechanism used and will involve a particular scheduling policy.

The behavior of a port consists of two main loops: one for receiving operation calls from external objects and storing these in the pending request table and one for dispatching calls from the pending request table and for forwarding these calls to an internal object and returning the result to the external object.

The first loop starts with a transition for each operation op that consists of a corresponding trigger

op(parameters)

where the list of parameters contains the caller and callee of the call. This trigger is followed by a local computation step which involves a corresponding update of the pending request table. After this local update the call to the operation op of the port-instance is completed. Note however, as described above, that the caller object, after completion of the call to the operation op of the port-instance will wait for a call to the operation return-op which will coincide with the completion of the operation op by the internal object.

The second loop starts with the selection of a call in the pending request table. The port-instance subsequently calls the corresponding operation of the callee. When this call is completed the port-instance resumes its activity by sending the received return value to the initial caller object.

5.5 Inter-components coordination

Reactive systems are systems that reacts continuously to their environment, at a speed imposed by the latter [HP85]. Among reactive systems are most of the industrial real-time systems, like control systems and signal processing systems. These systems are distributed in their own nature: think for example at the different location of the sensors and actuators of a system. Furthermore they are subject to temporal requirements concerning both the input rate and the response time. This requirements must be taken into account when modeling a system, for example, by considering an architectural design employing the *globally asynchronous locally synchronous* paradigm [Cha85]: communication within a unit of distribution may be synchronous, whereas communication between different unit of distribution must be asynchronous.

The semantic model of Ω-UML is rich enough to support communication through shared attributes, operation calls, and signals. Synchronous operation calls use rendez-vous as communication mechanism. It involves a synchronization between the sender and the receiver of the operation call for the message (and parameters) passing, an asynchronous establishment of the rendez-vouz, and another synchronization between the sender and receiver for passing the return values. Rendez-vous lead to useless waiting time and reduce parallelism and efficiency [Fox88]. That is why in this section we restrict the communication model of Ω-UML so to support the globally asynchronous locally synchronous paradigm: all inter-component communication are purely asynchronous (via signal events), while intra-component communication is unrestricted. This way, components are *units of distribution* of reactive real-time systems.

A consequence of the inter-component communication by signal events is that each port instance (the receivers of all the signals directed to a components) must be equipped with an event-queue, so that incoming requests are processed in a first-in-first-out fashion. This is equivalent to say that communication between components is performed by means of send and receive primitives over a network of FIFO channels: The processes in the network are the ports and the port instances. while the channels are the event-queue associated with each port instance. An asynchronous operation call $p.signal(parameters)$ correspond to sending the structured signal $signal(parameters)$ to the channel p (the identity of the port instance owning the event-queue), while the trigger of the signal correspond to the reception of the first signal from the channel with sink attached to the process. Notice that channels can be dynamically created, and passed to other processes by means of a signal. Therefore, communications are performed over a dynamically reconfigurable networks of channels and processes.

In other words, by loose coupling the inter-component communication mechanism (here obtained by forbidding synchronous operation call) one obtain a system of dynamic processes communicating through mobile channels. There is, however, an asymmetry in the above coordination mechanism: channels are mobile only at their source. This is due to the fact that in UML, the triggering of an operation is implicitly directed to the event queue of the active object controlling it. If we relax this constraint, and introduce trigger operations directed to a channel (introducing, for example, a syntax for trigger operation similar to a CSP [Hoa85] read operation c?signal(parameters)) then we obtain a the coordination model of mobile channels proposed in [ABdB00, SAdBB03]: Processes can be created dynamically and have an independent activity that proceeds in parallel with all the other processes in the system and interact only by sending and receiving messages *asynchronously* via channels which are (unbounded) FIFO buffers. Channels are created dynamically. In fact, the creation of a process consists of the creation of a channel which connects it with its creator. This channel has a unique identity which is initially known only to the created process and its creator. As with any channel, the identity of this initial channel too can be communicated to other processes via other channels, so that which processes are connected by which channels, is completely dynamic, without any regular structure imposed on it a priori.

A compositional formal semantics based on histories of signals sent and received by each process has been given for the above coordination model [dBB00], together with a logic-based component interface description language that conveys this observable semantics [AdBB00]. This interface description language allows for deriving properties of a component-based system out of the logical interfaces of each port of the constituent components [AdBB00]. Finally, the model has also been implemented as a middleware for distributed communication and collaboration [SAdBB02].

5.6 Conclusion

In this paper we have presented a UML model for components to address architecture and component based development. Components are units of abstraction that can be independently developed, like classes or modules. Unlike classes, they components are also unit of encapsulation that can be extended by subtyping of the interfaces, but not by inheritance of their implementation. Component-based systems are described by means of two new UML diagrams: component system diagrams and architectural diagrams. Component system diagrams are for describing the structural dependencies among the provided and required interfaces of the components in a system, while architectural diagrams are for the description of a runtime configurations of the architecture of a component-based system.

Our model offers a coherent view for the design of architecture and component-based systems: components serve as a naming mechanisms for abstracting from the internal parts, interfaces as declaration mechanisms of services (either provided or required) and ports together with the dependency-realization relations as abstraction mechanisms of object interactions.

Contrary to the component concept used in deployment diagrams of UML 1.4 [SP99], our components are not units of instantiation and do not need to have a unique run-time identity. Our model of component is similar to the recently approved proposal by the U2 partners for UML 2.0 [OMG03], but components have no state, are not instatiable, and allow for the existence at run time of multiple ports with the same set of interfaces, each Port attached to the necessary number of runtime links. For example, in UML 2.0 a class is also a component, while this is not the case for our notion of component.

Our model has been largely influenced by the main concepts offered by architecture description languages (ADLs): components, ports, and configurations. A large number of ADLs have been proposed, some of them with a sound formal foundation. We only mention here Wright [AG97], Rapide [LKA$^+$95] and ACME [GMW97]. Closer to our architectural diagrams are the architectural descriptions provided by ROOM [SGW94] and UML-RT [Sel98] (the latter is in fact a UML profile interpreting ROOM concepts in terms of UML stereotypes).

Many models for components have been proposed in the last years, some informal and remaining within the realm of the existing UML (see for example [CD00]), and others founded on a logical and mathematical basis (e.g. Broy's component model based on streams of messages [BS01]. In [GCK02] and [MRRR02], several strategies for modeling components and other architectural concepts within UML are investigated, with as conclusion that these concepts are hard to describe in UML as it is. Similar to Broy's component model, the semantics of our model is also based on sequences of messages (like those used for the semantics of CSP [Hoa85]). However our components have dynamic aspects (e.g. Port instances) not fully covered by Broy's model.

Moreover our component model is a conservative extension of an object-oriented model and therefore it requires the addition of only a couple of extra concepts to the standard UML 1.4. It is interesting to note that these additional concepts are also required by the component model proposed for UML 2.0 by the U2 partners [U20]. As described above, however, the semantics of these concepts is different between the two models.

Acknowledgement The work reported in this paper has been funded by the European IST-2001-33522 project OMEGA.

Chapter 6

The unified coordination language UnCL

Authors: Frank de Boer, Marcello Bonsangue, Joost Jacob

6.1 Introduction

In this paper we introduce a subset of UML called the Unified Coordination Language (UnCL) as an unified model for exogenous coordination. The main purpose of UnCL is to provide a separation of concerns between coordination and computation. In Fig. 6.1 we give an overview of our coordination model. UnCL provides a special class for every UML class (or group of UML classes) that is relevant for the coordination. This UnCL class imports operations (methods) of the application's class(es). It contains only attributes with references to objects within UnCL or to objects of the application.

In UnCL the additional coordination behavior is specified by associating state-machines with its classes. Such a state-machine in UnCL consists of transitions that involve two kinds of operations. The first kind are the application operations (*app-ops*) that are implemented (and executed) by the coordinated application. The second kind are coordination operations (*co-ops*). Objects in UnCL coordinate their interaction by means of these operations which are communicated via an event-queue system. Every UnCL object is associated with a queue which stores the messages involving its co-ops and that are sent to it.

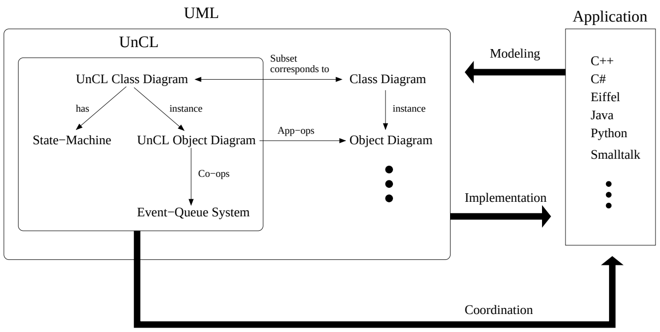
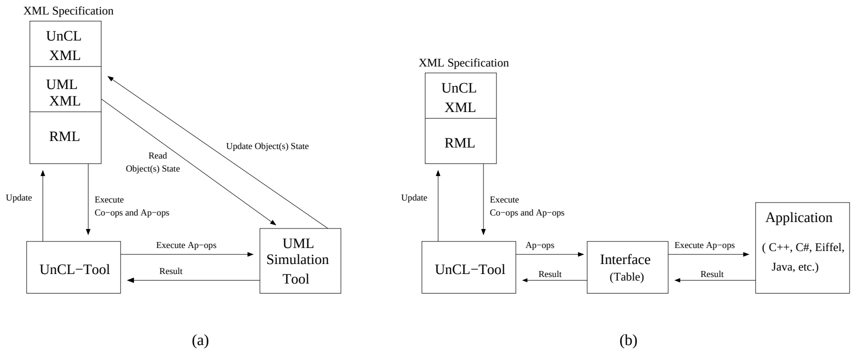


Figure 6.1: UnCL.

There are three dynamic aspects to UML models: (1)*app-op* semantics, (2) *co-op* semantics, and (3) *scheduling*; namely, which transition is to be fired at a precise moment. The computations inside an application involve (1), and the coordination involves aspects (2) and (3). In this paper we abstract away from the scheduling because we want to provide a separation of concerns between (2) and (3). This gives us the possibility to choose different scheduling algorithms for the same UnCL model. Thus, the scheduling semantics becomes a parameter of the model itself.



(a)

(b)

Figure 6.2: UnCL Tool Architecture.

In this paper we introduce a precise semantics of the coordination behavior of an UnCL model and its formalization in a new extension of XML [XML] called the *Rule Markup Language* (RML). RML is designed for the specification and execution of general transformations of XML data and is therefore very well suited for the specification and execution of the semantics of UML models. The application of RML to UnCL allows for both simulation within UML as well as the coordination of external applications at run-time.

In figure 6.2a we give the tool architecture for the simulation. The UnCL classes, state-machines, objects, and coordination mechanism are fully specified and given in XML. For every *co-op* a transformation rule in RML is given that describes how the UnCL XML specification of the input object-diagram changes by performing the operation. However, for executing an *app-op* the simulation needs a *UML simulation tool* that reads the relevant parts of the object-diagram, performs the operation, and changes the object diagram accordingly. The advantage of this architecture is that both the execution steps and the state of the UML application are fully given in XML, making it easy to see step-by-step how the state of the application evolves.

Using the same UnCL specification we can also coordinate run-time application(s). Such an application then basically serves as a kind of library whose objects are driven and coordinated by the UnCL model. Fig. 6.2b shows the corresponding coordination architecture. The attributes of UnCL classes now indirectly refer to run-time objects of the underlying application instead of UML objects specified by XML. The state of the application is hidden in the application itself and not part anymore of the XML specification. App-ops are now being performed by the run-time objects of the application. This means that we need an *interface* that binds the UnCL object references in XML to the run-time object references of the application. The interface maintains a table which relates these two different name spaces.

Plan of the paper: after describing UnCL in this section, we continue with the presentation of a precise semantics of UnCL. Then we discuss the execution platform and its main component RML. We finish with related work and conclusions.

6.2 Semantics of UnCL

In UnCL objects are coordinated by means of state machines. These state machines are associated with classes and consist of *transitions* of the form

$$l \xrightarrow{[g]t/a} l'$$

where l is the *entry* location and l' is the *exit* location of the transition. Furthermore, g denotes its boolean *guard*, t its *trigger*, and a its *action*.

More specifically, given a set of attributes, with typical element A, defined by the associated UnCL class the boolean guard g of a transition involves a call

$$A.op(A_1, \ldots, A_n)$$

to a boolean app-op *op* of the object denoted by A which is provided by the underlying application. We require that the execution of such a boolean app-op does not affect the values of the attributes defined by the UnCL diagram.

A trigger t is of the form

$$op(A_1, \ldots, A_n)$$

which specifies a co-op *op* defined by the UnCL class itself and a corresponding parameter list $A_1, \ldots, A_n$ of attributes.

Finally, an action involves a call

$$A.op(A_1, \ldots, A_n)$$

where *op* is either a co-op defined by the UnCL class of the object denoted by A or *op* is a app-op provided by its class of the underlying application.

We model object creation by means of a call $C.NEW(A)$ of an app-op *NEW* provided by the underlying application. This operation has a value/result parameter so the above call with actual parameter A will assign to A the identity of a new object in class C. In general we model assignments by means of value/result parameters, i.e., an assignment $B = A.op(A_1, \ldots, A_n)$ involving an operation-call is modeled by a call $A.op(B, A_1, \ldots, A_n)$ with value/result parameter B.

In order to formally define the *operational* semantics of state machines in UnCL we assume for each class c of a given UnCL class diagram a set O_c of *references* to objects in class c. In case class c extends c' (according to the UnCL diagram) we have that O_c is a subset of $O_{c'}$. (For classes which are not related by the inheritance hierarchy these sets are assumed to be disjoint.)

Definition 2 *An* object diagram *of a given UnCL class diagram with classes $c_1, \ldots, c_n$ can be specified mathematically by functions σ_c, for $c \in \{c_1, \ldots, c_n\}$, which specify for each object in class c existing in the object diagram the values of its attributes, i.e., $\sigma_c(o.A)$ denotes the value of attribute A of the object o, i.e., it denotes an object reference in $O_{c'}$, where c' is the (static) type of the attribute A (defined in the class c in the UnCL diagram).*

Often we omit the information about the class and write simply $\sigma(o.A)$. Control information of each object o in an object-diagram is given by $\sigma(o.L)$, assuming for each class an attribute L which is used to refer to the current location of the state machine of o. Furthermore, the event-queue of each object is given by the attribute E.

Given an UnCL class diagram consisting of a finite set of classes $c_1, \ldots, c_n$ and associated state machines, we define its behavior in terms of a *transition relation* on object diagrams. Object diagrams correspond to states in our semantic model. This transition relation is defined parametric in the semantics of the application operations and the way messages are stored and removed from the event-queue. More specifically, we assume for each action $a = A.op(A_1, \ldots, A_n)$ involving an app-op op a *labeled* transition relation

$$\sigma \xrightarrow{o.a} \sigma'$$

which specifies σ' as a possible result of the execution of the call a by the caller object o in σ. Such a labeled transition describes the *observable* effect on the UnCL object diagram of the execution of the corresponding call by the underlying application. As a special case we assume for each *guard* $g = A.op(A_1, \ldots, A_n)$ involving a boolean app-op op a *labeled* transition relation

$$\sigma \xrightarrow{o.g} b$$

where b denotes a boolean value which indicates the result of the operation (note that we assume that boolean operations does not affect the attributes of the UnCL diagram).

Furthermore, for each trigger $op(A_1, \ldots, A_n)$ we assume the semantic function

$$pop - op(A_1, \ldots, A_n))$$

which, given an input object diagram σ and an executing object o, returns the object diagram σ' that results from *removing* a message $op(o_1, \ldots, o_n)$ from the event-queue $\sigma(o.E)$ of o in σ and *assigning* the object references o_i to $\sigma(o.A_i)$, $i = 1, \ldots, n$, i.e., $\sigma'(o.A_i) = o_i$. In case there does not exist such a message this function is undefined.

On the other hand, given an input object diagram σ and a caller object o, the semantic function

$$push - op(A, A_1, \ldots, A_n)$$

returns the object diagram σ' that results from adding the message $op(o_1, \ldots, o_n)$ involving the co-op op sent by o to the event-queue $\sigma(o'.E)$ of the callee $o' = \sigma(o.A)$, where $o_i = \sigma(o.A_i)$, for $i = 1, \ldots, n$.

Definition 3 *Formally, given an UnCL class-diagram and the semantic interpretations of the app-op's, we have a transition $\sigma \rightarrow \sigma'$ from the object-diagram σ to the object-diagram σ' if the following holds: there exists an object o and a transition*

$$l \xrightarrow{[g]t/a} l'$$

in its state machine such that

Location $\sigma(o.L) = l$ *and* $\sigma'(o.L) = l'$*;*

Guard $\sigma \xrightarrow{o.g}$ *true;*

Trigger $pop - op(A_1, \ldots, A_n)(\sigma, o) = \sigma''$*, in case of a trigger* $t = op(A_1, \ldots, A_n)$*;*

Action *We distinguish between the following two cases:*

- *in case of a call $a = B.op(B_1, \ldots, B_k)$ involving a co-op op we have*

$$push - op(B, B_1, \ldots, B_k)(\sigma'', o) = \sigma'$$

- *in case of a call a involving an app-op we have*

$$\sigma'' \xrightarrow{o.a} \sigma'.$$

The first clause above describes the flow of control. The second clause states that the guard evaluates to true (without side-effects). The third clause describes the execution of the trigger by the executing object o in the initial object diagram σ in terms of the corresponding *pop-op* function. Note that the evaluation of the guard and the execution of the trigger are strictly sequentialized. This implies that the guard cannot refer to the new values of the actual parameters of the trigger which are stored in the event-queue. However, a slight modification would suffice to allow for this. For technical convenience only we restricted to a simpler semantic model. Finally, the execution of the action distinguishes between a co-op and an app-op. In both cases, the input diagram is the diagram resulting from the execution of the trigger and the diagram resulting from the execution of the action is the final result of the transition. A call to a co-op *op* is described in terms of the corresponding *push-op* which consists of pushing the message on the event-queue of the callee. Note that a call to a co-op is asynchronous and does not involve a rendez-vous with the callee. However such a synchronization can be modeled easily. Finally, a call to an app-op is described in terms of a corresponding labeled transition which models the execution of the call by the underlying application.

Note that the execution of a transition of a state-machine is atomic. However, more fine-grained modes of execution can be introduced in a straightforward manner.

6.3 The UnCL Execution Platform

The kernel of the UnCL tool consists of an algorithm for taking transitions in the state-machine, scheduling the transitions, calling the coordinated application and managing a user interface. The part of the algorithm that concerns the coordination, i.e., the processing of the co-op's, is defined using RML rules.

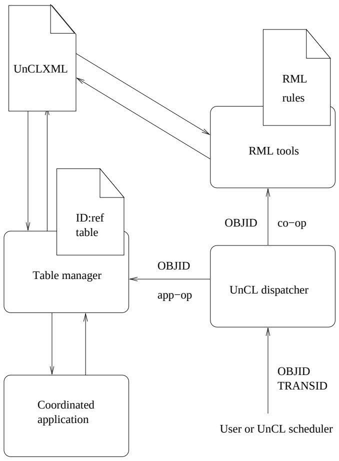


Figure 6.3: RML in UnCL

Fig. 6.3 shows how the RML tools are combined with the UnCL tool, the XML for UnCL models in a new XML vocabulary called UnCLXML, and the coordinated application. A RelaxNG schema for UnCLXML is available at [Jacb]. What is not displayed is a scheduler that decides what transitions to take at a specific moment, this scheduler can be implemented independently, useful for real-time applications, or the scheduler can be a human being using a UnCL system as the web-application at [Jacb]. In the Figure we see a document icon for the UnCLXML document that is publicly visible. We see two other document icons for the RML rules, visible only to the RML tools part of UnCL, and the table relating object references in XML to the run-time object references of the application, visible only to the UnCL table manager that has to be implemented in the same language as the coordinated application. Both the table manager and the RML tools can modify the UnCLXML document, so the document has to be protected by a locking mechanism. The UnCL dispatcher receives an object identifier (OBJID) and a transition identifier (TRANSID) in the case of more than one possible transition for the object, and the dispatcher sends the necessary information to the RML tools in the case of a co-op and to the Table manager in the case of an app-op.

In the following we will show how state-machines and their semantics as defined in Sect. 6.2 can be encoded in the Rule Markup Language (RML) which is a new extension of XML for specifying and executing XML data. RML can be combined with an XMLvoc in order to define transformations of corresponding XML data using the XMLvoc itself. In the case of UnCL we combine RML with UnCLXML. With RML we can now define also the semantics of UML models in XML. Furthermore, using the RML tool we can *execute* these models. UnCL *users* do **not** have to learn RML, but just write state machines in UnCLXML.

The rules defined with RML consist of an *antecedent* (the input) and a *consequence* (the output). The antecedent and the consequence consist of XML+RML, where XML is the problem domain vocabulary and RML is used to specify *RML-variables* in the XML. The antecedent of a rule will be *matched* with input XML, resulting in a binding for all the RML variables to corresponding XML constructs in the input XML. These constructs can be XML element names, XML attribute names, XML attribute values, whole XML elements including their children, or sequences of XML elements with their children. If a match of the input XML with the antecedent of a rule is possible then there will be a specific XML element in the input XML that matches the antecedent, and this XML element will be *replaced* with the consequence of the rule. The consequence of a rule also contains XML+RML.

The RML-variable names will be replaced with the corresponding contents of the RML variables in the output.

The table in Fig. 2.1 sums up all current RML elements *with a short description of their usage*. Due to a lack of space we have to refer to the RML tutorial in Chapter 3 for a longer description of all the RML elements. It is easy to think of many more useful elements in RML than appear in the table. Not everything imaginable is implemented because a design goal of RML is to keep it as concise and elegant as possible. Only constructs that have proven themselves useful in practice are added.

Variable binding of RML-variables during the matching of the antecedent of a rule is attempted in the order of the elements in the input XML tree. If an input XML tree contains more than one match for a variable then only the first match is used for a transformation. If you want to transform all matches then you will have to repeat applying the rule on the input.

Binding of RML-variables can also be done *before* a rule is applied if the RML-variables are supposed to contain string values; in that case the matching will only succeed if the supplied string values appear in the input XML in the position where the RML variable appears. An example of this pre-binding of variables in the UnCL tool is when the user supplies an object ID (variable IDOBJ in the examples that follow later) when the user wants that object to take a transition. To pre-bind a value of `id002` to RML variable `IDOBJ`, the user can supply an extra argument for the RML tools: `IDOBJ=id002`. Such pre-binding can also be done when using the RML libraries instead of the command-line tools.

The RML tutorial also describes a concise XML vocabulary for defining RML recipes, called Recipe RML (RRML). RRML is used to define *sequences* of, possibly iterated, transformations and has proven itself useful in alleviating the need for writing shell scripts or functions in a programming language containing sequences of calls to the RML tools. The idea is to avoid programming and to define as much as possible in XML in a data driven design.

Figure 6.4 shows a simple example RML rule. This is a rule that is used after a transition has been taken successfully by an object modeled with UnCL. With this rule the `location` attribute of the object is assigned the value of the `target` attribute. An example of the effect of the rule would be that

```
...
<obj id="id538" location="state_3" target="state_5" ... >
  <queue>
    ...
  </queue>
</obj>
...
```

is changed into

```
...
<obj id="id538" location="state_5" target="None" ... >
  <queue>
    ...
  </queue>
</obj>
...
```

for an object with identifier `id538`.

```
<div class="rule" name="set location">
  <div class="antecedent">
    <obj id="rml-IDOBJ" location="rml-L" target="rml-T"
         rml-others="rml-O" >
      <rml-list name="ObjChildren"/>
    </obj>
  </div>
  <div class="consequence">
    <obj id="rml-IDOBJ" location="rml-T" target="None"
         rml-others="rml-O">
      <rml-use name="ObjChildren"/>
    </obj>
  </div>
</div>
```

Figure 6.4: The example RML rule

When applying this rule, the RML transformation tool first searches for an `obj` element in the input, corresponding with the `obj` element in the `antecedent` of the rule. These `obj` elements match if the `obj` in the input has an `id` attribute with the value bound to the RML `IDOBJ` variable mentioned in the antecedent, in the example this value is `id538` and it is bound to the RML variable `IDOBJ` *before* the rule is applied. This pre-binding of some of the variables is how UnCL can manage and schedule the execution of the RML transformation rules. If the `obj` elements match, then the other RML variables (`L`, `T`, `O` and `ObjChildren`) are filled with variables from the input `obj`. The `L`, `T` and `O` variables are bound to strings, the `ObjChildren` variable is bound to the children of the `obj` element: a list of elements and all their children. The `consequence` of the rule creates a new `obj` element, using

```
<div class="rule">
  <div class="antecedent">
    <UnCL>
      <classdiagram>
        ...
        <class name="rml-ClassName">
          ...
          <statemachine>
            ...
            <transition id="rml-IDTRANS">
              ...
              <trigger>
                <op name="rml-TriggerName">
                  <rml-list name="Params"/>
                </op>
              </trigger>
              ...
            </transition>
            ...
          </statemachine>
        </class>
        ...
      </classdiagram>
      <objectdiagram>
        ...
        <obj class="rml-ClassName"
             id="rml-IDOBJ"
             rml-others="rml-OtherObjAttrs">
          ...
          <queue>
            <rml-list name="PreEvents"/>
            <op name="rml-TriggerName"/>
            <rml-list name="PostEvents"/>
          </queue>
        </obj>
        ...
      </objectdiagram>
    </UnCL>
  </div>
```

```
  <div class="consequence">
    <UnCL>
      <classdiagram>
        ...
        <class name="rml-ClassName">
          ...
          <statemachine>
            ...
            <transition id="rml-IDTRANS">
              ...
              <trigger>
                <op name="rml-TriggerName">
                  <rml-use name="Params"/>
                </op>
              </trigger>
              ...
            </transition>
            ...
          </statemachine>
        </class>
        ...
      </classdiagram>
      <objectdiagram>
        ...
        <obj class="rml-ClassName"
             id="rml-IDOBJ"
             rml-others="rml-OtherObjAttrs">
          ...
          <queue>
            <rml-use name="PreEvents"/>
            <rml-use name="PostEvents"/>
          </queue>
        </obj>
        ...
      </objectdiagram>
    </UnCL>
  </div>
</div>
```

Figure 6.5: RML rule for removing an event from the event-queue

the values bound to the RML variables, and *replaces* the `obj` element in the input with this new `obj` element.

Due to lack of space we restrict the description of the formalization in RML of the processing of the co-op's to the removal of a message from the event-queue, as shown in Fig. 6.5. The figure contains some lines with `...` in places where `rml-list` and `rml-use` constructs are used to preserve input context in the output. Here we see that in RML a pattern can be matched that is distributed over remote parts in the XML, the remoteness of the parts is why the rule has so many lines. In short, this rule looks for the name of the trigger that indicates the message that has to be removed from the event-queue, and then simply copies the event-queue without that event. But to find that name of the trigger, a search through the whole UnCLXML model has to take place, involving the following steps.

During application of this rule, the matching algorithm first tries to match the input with the antecedent of the rule, where IDOBJ and IDTRANS are pre-bound RML variables. With these pre-bound variables it can find the correct obj, then it finds the ClassName for that object. With the ClassName the class of the object can be found in the classdiagram in UnCLXML. When the class of the object is found, the transition in that class with id TRANSID can be found and in that transition element in the input we can finally find the desired TriggerName. The algorithm then looks for a message with name TriggerName in the event-queue of the obj, and binds all other events in the event-queue to RML variables PreEvents and PostEvents. In the consequence of the rule then, all these bound RML variables are available to produce a copy of the input, with the exception that the correct event is removed.

6.4 UnCL and Mobile Channels

In UnCL state machines model communication between objects in terms of the coordination operations which involve a simple event-queue mechanism. This provides a separation of concerns between the computational part specified by the application and the coordination part specified by UnCL. Due to this separation of concerns it is possible to replace the event-queue mechanism with any other coordination mechanism. Preferably, with one that preserves the separation of concerns and is easy to implement in concurrent and distributed systems. An example of such coordination mechanisms are shared data spaces like Linda [CG90] and JavaSpaces [EFA90]. In this section we discuss the replacement of the event-queue by another coordination mechanism called *MoCha* [SAdBB03]. MoCha is an exogenous coordination framework for (distributed) communication and collaboration using *mobile channels* as its medium.

6.4.1 MoCha's Mobile Channels

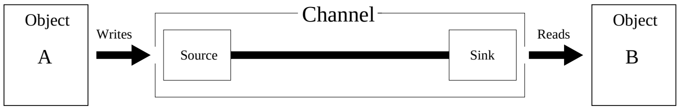


Figure 6.6: General View of a Channel.

A channel in MoCha, see figure 6.6, consists of two distinct ends: usually (*source*, *sink*) for most common channel-types, but also (*source*, *source*) and (*sink*, *sink*) for special types. These channel-ends are available to the objects of an application. Objects can *write* by inserting values to the source-end, and *read* by removing values from the sink-end of a channel; the data-flow is locally *one way*: from an object into a channel or from a channel into an object.

Channels are *point-to-point*, they provide a directed virtual path between the (remote) objects involved in the connection. Therefore, using channels to express the communication carried out within an application is *architecturally very expressive*, because it is easy to see which objects (potentially) exchange data with each other. This makes it easier to apply tools for dependencies and data-flow analysis of an application.

Channels provide *anonymous connections*. This enables objects to exchange messages with other objects without having to know *where* in the network those other objects reside, *who* produces and consumes the exchanged messages, and *when* a particular message was produced or will be consumed. Since the objects do not know each other, it is easy to update or exchange any one of them without the knowledge of the object at the other side of the channel. This provides objects that are loosely coupled in space and time.

The ends of a channel are *mobile*. We introduce here two definitions of mobility: logical and physical. The first is defined as the property of passing on channel-end identities through channels themselves to other objects in the application; spreading the knowledge of channel-ends references by means of channels. The second is defined as physically moving a channel-end from one location to another location in a distributed system, where location is a *logical address space* where objects execute. Both kinds of mobility are supported by MoCha.

Because the communication via channels is also *anonymous*, when a channel-end moves, the object at the other side of the channel is not aware nor affected by this movement. Mobility allows dynamic reconfiguration of channel connections among the objects in an application, a property that is very useful and even crucial in systems where objects are mobile. An object is called mobile when, in a distributed system, it can move from one location (where its code is executing) to another.

Channels provide transparent *exogenous coordination*. Channels allow several different types of connections among objects without them knowing which channel types they are dealing with. Only the creator of the connection knows the type of the channel, which is either synchronous or asynchronous. This makes it possible to coordinate objects from the 'outside' (exogenous), and, thus, change the application's behavior without having to change the code of it's classes.

6.4.2 Channel Types

MoCha supports eleven types of channels. All with the same interface, but with different behavior. We give a short description of three major channel types. For more details and the remaining channel types we refer to the MoCha middleware manual [SAdBB02].

- *Synchronous channel.* The I/O operations on the two ends are synchronized. A *write* on the source-end can succeed only when a *take* operation also atomically succeeds on the sink-end, and vice-versa. A *take* operation is the destructive version of the *read* operation.

- *Lossy synchronous channel.* If there is no I/O operation performed on the sink channel-end while writing a value to the source-end, then the *write* operation always succeeds but the value gets lost. In all other cases, the channel behaves like a normal synchronous type.

- *Asynchronous unbounded FIFO channel.* The I/O operations performed on the two channel-ends succeeds asynchronously. Values written into the source channel-end are stored in the channel in a FIFO distributed buffer until taken from the sink-end.

6.4.3 Implementation

The MoCha framework is implemented in the *Java* language using the *Remote Method Invocation package* (RMI). This MoCha middleware can be used for both distributed and non-distributed applications. The middleware has a clear and easy high-level application programming interface (API).

6.4.4 UnCL and MoCha

Replacing the event-queues by MoCha channels requires the introduction of channel-ends in UnCL and the definition of their coordination operations in the state-machine semantics. Since channel-ends are also UML classes, we accomplish the first, by allowing the UnCL class attributes to also refer to these channel-end objects. The state-machine coordination operations are defined as:

- $T.new(L, E_1, E_2)$ creates a new channel, where $\{E_1, E_2\}$ are attributes storing the created channel-ends. T is an attribute that refers to the type of the channel, and L is an attribute that refers to a particular location. In the MoCha middleware such a creation is translated into the expression `chan = new MobileChannel(L,T)`, where `chan.E1` and `chan.E2` are the attributes that refer to the ends of the new created channel.

- $E.write(V)$ writes the reference value of attribute V to the source channel-end E.

- $E.take(V)$ takes a reference from the sink channel-end E and stores it in attribute V.

- $E.read(V)$ reads a reference from the sink channel-end E and stores it in attribute V. (read is the non-destructive version of *take*).

- $E.move(L)$ moves a channel-end E to location L.

Observe that, in cases where we are not concerned with modeling locations we can take the same first four operations and remove the location attribute L.

Using MoCha in UnCL has four major advantages. First, since the MoCha framework is implemented in the Java language, there is a straightforward implementation for every UnCL model. Straightforward in the sense that the MoCha middleware implements the same operations and channels as the ones of the UnCL + MoCha model, providing a one-to-one relation between a UnCL channel and a MoCha middleware channel. Second, since MoCha supports distributed environments, every UnCL model automatically does as well. Third, the UnCL model now provides the means for more high-level exogenous coordination. In addition of changing the state-machines, with MoCha we can also change the application's behavior by simple choosing a different type of channel between objects. And finally four, MoCha enhances the, already present, separation of concerns between the computational part and the coordination part of an application.

6.5 Conclusions and Related Work

In this paper we presented an *Unified Coordination Language* (UnCL) that is based on a separation of concerns between coordination and computation. UnCL provides a general language for coordination given in UML that can be used both for simulation and coordination of an application at run-time. We discussed a precise semantics of UnCL state machines, the UnCL execution platform, and how to use an executable extension of XML specifications within this platform. Finally, we discussed the possibility of incorporating MoCha into UnCL.

UnCL relates to other coordination languages like *Linda* [CG90], *JavaSpaces* [EFA90], and MANIFOLD [Arb96]. For the majority of these models UML interfaces are made. However, as far as we know, UnCL is the first coordination language that fully integrates with UML. Besides modeling coordination a UnCL UML-specification can also coordinate an application in runtime. UnCL + MoCha relates to Reo[Arb04], an exogenous coordination language where complex channel connections are compositionally build out of simpler ones.

Other related work on coordination modeling are SOCCA [EG94] and CSP-OZ [MORW04]. SOCCA is an object-oriented specification language supporting the arbitrarily fine-grained synchronization of processes. Despite the fact that SOCCA is related to UML it is a separate language and not an extension like UnCL. CSP-OZ is an integrated formal method combining the process algebra CSP with the specification language Object-Z. It provides the means for putting special information (tags) in UML class diagrams. The full CSP-OZ specification is obtained after compiling these class diagrams, unlike UnCL where the specification is fully given in UML. Both CSP-OZ and UnCL + MoCha use channels as the coordination mechanisms. However, CSP-OZ channels are static while UnCL + MoCha channels are dynamic. This enables UnCL to specify dynamic reconfigurable coordination patterns.

In our approach we abstract away from a particular scheduling algorithm. This gives us the advantage to make such an algorithm a parameter of an UnCL model. This is different from other work like [vdB01], [LMM99], and [PL99] where scheduling is already integrated into the semantics, making it more difficult to change the already present scheduling algorithm (if desired).

Instead of using RML for the UnCL transformation rules we could have used other tools for XML transformations, like XSLT [Cla]. We chose RML because it was developed with more complex matching patterns in mind: The

XML wild-cards defined with RML can be distributed over several places in the input. Such a distributed matching pattern is hard to define with XSLT, because XSLT templates are path oriented instead of pattern oriented.

We have successfully used the UnCL architecture in project OMEGA IST-2001-33522, sponsored by the European Commission, where we formalized the OMEGA subset of UML and will apply it to industrial case studies. A first test case is demonstrated on-line at [Jacb].

Chapter 7

ATL Applied to the Tableau Method

Author: Joost Jacob

7.1 Introduction

We have created a transformation language called ATL[1] that can be used to define facts and rules in a convenient way that is also suitable for non–programmers. We show how it can be used in a semantic tableau method to generate proofs, where the facts define axioms and the rules define theories. To do this, we create executable functions from ATL rules and these functions are then used in the implementation of a tool for the semantic tableau method. The tableau method we use is extended with equivalence classes in order to provide automatic unification. The resulting proof system is powerful but still transparant and easy to use since the axioms and theories are defined in the user's own notation. Although full automation *is* possible, the prime goal of the proof system is not sophisticated automation of proof, but rather to make it convenient for the user to define axioms and theorems and to help with routine unification of equalities, in order to arrive at a kind of proofs where the emphasis is on elegance and where a high level of abstraction

[1] ATL stands for ASCII Transformation Language, intended for manipulations of symbols that can be formed with an ASCII keyboard. The ASCII aspect is now outdated since also Unicode is supported, but the name stuck.

is encouraged. Such proofs can generally not be created automatically, the human contribution in the design of the proof is very significant. There is a growing demand for such proofs, because they are often easier to understand and thus more convincing[2] than fully automatically generated versions, and because they often help understand the subject matter better.

To demonstrate the above, and to provide a good motivation for the work in this paper, we follow an example that we encountered in the OMEGA[OME] project where we proof a property of a software model. Software is often *in*correct and using a well–known prover like PVS [ORR$^+$96], as we did in OMEGA, only works well for a correct model. What to do if it is not correct? The tableau method, looking for a contradiction, seemed especially suitable for, abstract, high–level, software model verification.

We did investigate literature and the internet to see if we could find a tableau tool that we could use but we were unsuccessful. Our requirements for such a tool were that it would not have a steep learning curve, without for example having to learn a functional programming language like with ACL2 [KMM00], and we wished to be free in syntax notation and the tool should be preferably independent of an operating system, versioning problems, or software libraries. That is the reason we started this work. We have created a web–application, wherein the user can define rules in ATL, that can be used to derive the proof mentioned above. The development of ATL did help considerably to make it possible to create the web–application on schedule and in time and to make it easy to use. The tool also uses the Python [vR95] programming language that generates HTML for presenting the user with forms to fill in and that performs various bookkeeping tasks and is also used for the implementation of unification. We have plans to develop a scripting language that can be combined with ATL, or is an extension, to replace some of the Python software. An ATL interpreter is already available that could in *theory* already -do this, but we need to develop a very high–level language so the user can more easily define for instance proof search strategies. As it is now, the tool takes one derivation step at a time and the user is the scheduler: the user has to click on a button with a rule–name to choose a step. The web–application and the ATL tools are publicly accessible from the URL given in Sect. 7.3. In the web–application the user can derive a proof in the sandbox, or a new project can be started with new notation, new axioms and new theories.

[2] especially for the executive kind of persons

Overview of the next Sections

In the next Section we introduce ATL and we give a language theoretic basis, and an operational semantics for a new reduction rule that we need in Sect. 7.2.4. Following that is a Section about our tableau method. We demonstrate how the α– and β–rules from the semantic tableau method are defined in ATL and how the user can add new rules in possibly new notation. We also discuss the addition of equivalence classes to the tableau to add considerable unification power to the tableau method. Section 7.4 shows how we prove a property of a software model with the tableau. Section 7.5 is a conclusion with some related and future work.

7.2 ASCII Transformation Language (ATL)

The design goal for ATL was to be able to turn derivation rules for our tableau into executable functions, in a general and convenient way. An important aspect of the convenience is syntax independence, the ability to handle user–defined notation. Suppose for example that a user defines the $\alpha4$ semantic tableau rule in the following syntax

```
~(X -> Y)
    |
   X,~Y
```

where it is clear for a human reader what the user means, especially for a reader that is familiar with semantic tableaux, but for a computer program it is not clear. The ”`~`” stands for *not*, the ”`->`” for implication, and the comma ”`,`” for *and*, or in our case rather the separator between two sentences in a branch in the tableau.

Let us forget for a moment the meaning of this rule, and view this rule as a string–rewrite. We note that the `X` and `Y` are like *wildcard characters*. A wildcard character can be used to substitute for any other character or characters in a string[3]. The rule accepts string of the form `~(+ -> +)` where the `+` is a wildcard for one or more characters. If we could *name* the wildcards, remembering the characters they substitute, we can re–use them in the output of the rule. What we need then, is a way to distinguish wildcards

[3] This definition is from FS 1037C, a Telecommunications standard, at `http://www.its.bldrdoc.gov/fs-1037/fs-1037c.htm`

from the constant strings, and a way to name them. This idea is explained in more detail and more formal in the rest of this Section, for an early intuitive understanding we give the $\alpha 4$ rule here in ATL:

```
~(var:i -> var:j)
=def
var:i,~(var:j)
```

where `var:` is used to denote a wildcard, and the name that follows immediately is the name of the wildcard. The `=def` is used to separate rule input from rule output. This ATL encoding of the $\alpha 4$ rule can be input into an ATL function that accepts as another argument an input string, and that produces the desired transformation if the input string matches the `~(+ -> +)` pattern. The definition of this $\alpha 4$ rule is convenient for the user since the user can use an own notation of choice. The meaning of the various symbols in the notation remains the responsibility of the user. This approach is different from the approach taken by tools that enforce a notation on the user or tools that enforce a type system. We believe that our approach is more flexible, and can still be extended to the other approaches.

We have created a transformation language, called ATL, that extends the λ–calculus [Chu41] with pattern matching. The λ–calculus contains two reduction rules and our extension consists of one extra reduction rule. With the extra reduction rule it becomes practical to define transformations such as are desired for the rule–based string transformations that we need to define axioms and theorems in our tableau. The λ–calculus is Turing complete, and ATL can mimic the λ–calculus by using only rules that correspond to λ–expressions, with patterns that match only one variable, thereby turning the extra reduction rule in ATL into a dummy transformation. This means that ATL also is a universal model of computation, Turing complete, and leads to a new computational model, but that is out of scope for this paper, however what is interesting here is that as a consequence we are assured that every theorem is expressable in ATL.

7.2.1 Preliminary: λ–calculus

In λ–calculus we have functions that accept input and produce output. The λ–calculus uses the well–known notation with the λ character to distinguish a function expression from an ordinary expression that is the result of applying the function.

Suppose we have a painting function that produces the following outputs for the corresponding inputs:

```
red -> painted red
blue -> painted blue
green -> painted green
```

We can use λ-calculus to describe such a function: `Lx.painted x`. This is called a lambda–abstraction, a kind of lambda–expression. The other kind of lambda–expressions are identifiers and function application. The `L` is supposed to be a lowercase Greek λ character.[4]

We can identify a function with a name: `doPaint === Lx.painted x`. *Application* of a function to an argument is written as:

```
(doPaint red) -> painted red
```

where application of function `f` to argument `x` is written as `(f x)`, like in the LISP programming language. Identifying a function with a name is a useful abstraction, but note that this abstraction is not an official part of the λ–calculus where every function is anonymous.

The evaluation of a λ–expression is from the application of two reduction rules.

The α–reduction rule

The *α–reduction rule* says that we can consistently rename bindings of variables:

```
Lx.E -> Lz.E[z/x]
```

for any z which is neither free nor bound in E, where `E[z/x]` means the substitution of `z` for `x` for any free occurance of `x` in `E`.

[4] We do not use a real λ character here because it is difficult to get it into the *verbatim* character set that we want to use to show "code" examples. As such, the λ character itself can be considered a disadvantage of the λ–calculus since it makes it harder to use in typical source editors. It exemplifies the fact that λ–calculus was not meant for programming, it has mainly theoretical purposes.

The β–reduction rule

The *β–reduction rule* says that application of a λ–expression to an argument is the consistent replacement of the argument for the λ–expression's bound variable in its body:

```
(Lx.E)Q -> E[Q/x]
```

where `E[Q/x]` means the substitution of `Q` for `x` for any free occurrance of `x` in `E`. The Church–Rosser Theorem states that the final result of a chain of substitutions does not depend on the order in which the substitutions are performed.

7.2.2 ATL

- Like in λ–calculus, in ATL we also have functions that accept input and produce output, we call them rules. The input for a rule is a string, and the output of a rule is also a string.

The λ-expression `Lx.2x` is written in ATL as the rule `var:x =def 2var:x`. Everything before the `=def` we call the *antecedent* of the rule, everything after it we call the *consequent* of the rule. Roughly comparing ATL with λ-calculus, the rule antecedent maps to "λ", and the `=def` maps to the "`.`".[5]

The input for a function in λ–calculus is exactly one argument, whereas the input for an ATL rule is a string, and from this string we can extract more than one argument. The string is matched with a pattern (the rule antecedent) that extracts the arguments from the input string and binds them to variable names, like names of function formal parameters. ATL applies the language–theoretic idea of regular expressions in its design; the matching pattern is a template with variables. The consequent of an ATL rule is also a pattern like that in the antecedent, this pattern is not used for matching but for construction of the output: the variables are replaced with their bindings from the match.

For a function expression in ATL that corresponds with "`Lx.painted x`" in the paint example we write `var:x =def painted var:x` where we see

[5] In the example the string 2 is used and as such it does not mean the number 2, but it *does* after we have identified the string 2 with the Church integer, see [Chu41], defined as `Lf.(Lx.(f (f x)))`, a higher–order function that takes a function f as argument and returns the 2–fold composition $f \circ f$.

the `var:` notation to denote variables that together with the `=def` notation is all that we need to define rules. Every variable name start with `var:` and ends with a name-string, where we define the name—string like an XML–name[6].

To the α– and β–reduction rules in λ–calculus we add a γ reduction rule that reduces a *rule application* to a *consequent application*. Where the λ–calculus applies the β rule, in ATL we have to apply the γ rule first, so we have a two–step reduction instead of one step:

```
    ((A =def C) inputstring) -> (C frame) -> output
```

where `A =def C` is the rule and `C` is the consequent of the rule. The matching of the antecedent creates a frame[7], a set of name–value pairs, that provides a binding for the variables. The `frame` dictionary is created from the `inputstring` via a matching algorithm described below. We write a frame between curly braces like {`var:x=foo, var:y=bar`} where the name `var:x` is bound to `foo` and the name `var:y` is bound to `bar`.

We will now first show how the λ–calculus paint example is expressed in ATL. The γ rule applied to the example:

```
((var:x =def painted var:x) red) ->
                                    ((painted var:x) {var:x=red)
```

removes the antecedent and the `=def` and builds a frame from the input wherein variable `var:x` is bound to `red`.

For application of the ATL consequent we use the same parenthesis–syntax as in λ–calculus, except that the input is now a frame. The application of the consequent goes via simple substitution like in the β—reduction from λ-calculus:

```
    ((painted var:x) {var:x=red}) -> painted red
```

In the given example the creation of a frame in the γ step looks pointless since there is only one binding in the frame and the whole input string is the value. To see why the two–step reduction is useful, suppose that our

[6] See the XML Specification at `http://www.w3.org/TR/REC-xml`. The choice for the XML name definition is because of the Unicode support in XML, which is used in other work were we combine ATL with XML transformations. For the purpose of this paper consider name–strings just to be strings that start with a letter

[7] In this Section we use the concepts of a **frame** from [AwJS96]

input is not just the string ”`red`” but the string ”`bg=blue fg=red`”. This is a common situation, the longer string could for example be attributes of an XML element that define foreground and background colors. This situation can be handled with the rule

```
var:y fg=var:x
=def
painted var:x
```

The γ and β rule applications

```
((var:y fg=var:x =def painted var:x) bg=blue fg=red)
->
((painted var:x) {var:y=bg=blue,var:x=red})
->
painted red
```

now also result in ”`painted red`”. The γ–rule matches ”`var:y`” with ”`bg=blue`” and ”`var:x`” with ”`red`”, creating the frame {`var:y=bg=blue var:x=red`}. The ”`fg=`” substring from the input is discarded. By adapting the antecedent we extract the desired ”`red`” value from the input and bind it to `var:x`. We leave the definition of the rule consequent unchanged.

We give another example of the usefulness of the two–step reduction. The function $f(x, y) = x + y$ is written in λ–calculus as `Lx.Ly.x + y`, a higher—order function of one argument that returns a function of one argument. The γ–reduction step in ATL can eliminate some need for higher–order, this function is written more readable as ”var:x var:y =def var:x + var:y”, expecting an inputstring with x and y separated by whitespace. A direct translation of the λ–expression is also possible in ATL, but not recommended: ”var:x =def var:y =def var:x + var:y”. This one first accepts an inputstring with the value for x, and then generates a new ATL rule with that value. The new rule accepts the value for y and returns `x + y`. For the definition of axioms and theorems in our tableau method we avoid higher–order functions. An example rule for AND elimination can be defined as simple as ”var:x AND var:x =def var:x”, without the need for higher–order functions and with the additional benefit of constraining valid input to two equal strings separated by an `AND` string. It was this kind of rules that ATL was designed for in the first place.

7.2.3 Implementation

A library is available[8] that contains a function that takes an inpu-t-string and an ATL rule as arguments, and returns the output–string.

7.2.4 Definition of the γ–reduction

The γ—reduction maps a *rule* application with an *inputstring* parameter to a *consequent* application with a $frame$ parameter:

`(rule inputstring)` $\mapsto$ `(consequent frame)`

The rule is a string that contains the substring `=def`. The *antecedent* of the rule is everything before the `=def` and the *consequent* is everything after it, so the production of *consequent* is a simple string tail extraction. The γ–reduction results in a (possibly empty) frame if the *inputstring* is an element of the set of strings defined by the *pattern* formed by the *antecedent*. A *pattern* is an ordered list of interleaved constant strings $cs_{1..n}$ and named wildcards $v_{1..k}$ and it defines the (infinite) set of strings that can be formed by substituting every wildcard by any string. If a wildcard occurs at more than one place in the pattern, i.e., with the same name, then it has to be substituted by the same string. A wildcard v_j, with $1 \leq j \leq k$, that is substituted by a substring s of *inputstring* adds the pair $(name, s)$ to the set $frame$, where *name* is the name of the wildcard. If the substring s contains parens then it must be well–formed, i.e., all opening parens must be closed. This is an important constraint on valid bindings, it makes it possible for the user to disambiguate rules when necessary[9]. This well–formedness constraint holds also for braces and square brackets in the current ATL implementation. The tool can be configured to add a well–formedness constraint for angle brackets, or to remove for example the constraint for braces. If there is more than one way to match *inputstring* with *pattern* in this way, then we take for every v_i, with $1 \leq i \leq k$, the shortest possible match before matching

[8] Library `atl` at `http://homepages.cwi.nl/~jacob/atl/`. The mentioned function is called `transform`. Python was chosen because Python is very interoperable with other languages, Python code can for example be translated to Java byte code. The `atl.transform` function is used in our semantic tableau web–application.

[9] Consider the difference between matching "a and b and c" and "(a and b) and c" with the pattern "var:x and var:y". In the first case x will be bound to `a`, in the second case x will be bound to `(a and b)`, because `(a`, being the first match found from left–to–right, is not well–formed and therefore rejected.

v_{i+1}. If *inputstring* is not an element of the set defined by *antecedent*, then the γ–reduction returns the unchanged *rule* and inputstring.

7.3 A webapplication

In this Section we introduce the tableau method as we use it and we introduce the web–application that can be used for the method. The web–application is publicly available at `http://homepages.cwi.nl/~jacob/st/index.html`. Because it is a web–application, a user does not have to download software but always has the latest version via a browser, and browser–features like the "Back"-button are available to redo steps in a derivation, making it suitable to iteratively develop a proof. We will explain by example how to define derivation rules with ATL and how our application handles them. First we show how the standard α– and β—rule are implemented and then how the user can add his or her own rules in a notation of choice.

For the notation of the α– and β–rule we had to decide on a notation for propositional logic, it is:

`~`	`NOT`
`&`	`AND`
`v`	`OR`
`->`	`IMPLIES`
`<->`	`IF AND ONLY IF`

We did choose a text notation because the web–application has a HTML Textarea that contains the derivation tree and we want the user to be able to edit the derivation tree, so proof derivations can be varied and retried in an easier way than would be the case if we choose real mathematic symbols in a more comples user interface.

The webapplicaton consists of password–protected projects where each project has the standard α– and β–rules predefined and where new rules and notation can be added. An interested user could also experiment with a redefinition of the α– and β–rules in his own notation if so desired, but sticking to the predefined notation of choice is of course less work. There is also one project without password that is called the Sandbox.

An example of the α–4 rule as implemented in ATL was already shown in Section 7.2. If you skipped Section 7.2 but are familiar with wildcard matching, a common technique in for example shell programming, then look at α–4 example there, the rule can be understood by `var:` being prefix–notation for a wildcard name and `=def` being a separator between input and output.

We now show how to implement a β–rule in ATL, where branching in the derivation tree occurs. The derivation tree in the textarea consists of lines with sentences, where every line is a branch in the derivation tree. This representation has the advantage that every line can now be worked on independently, a disadvantage is that in the case of a β–rule applied to a sentence, the other sentences in the branch have to be copied to a new line. In our work we found that the advantage outweighs the disadvantage, but of course this depends on the kind of proofs that you want to derive.

Beta–rule 2 rewrites implication, creating a new branch:

```
X -> Y
  / \
~X   Y
```

In ATL this rule is implemented with

```
var:i -> var:j
=def
~var:i
var:j
```

changing a one–line sentence to two lines. The web–application copies all the other sentences in a branch if a rule creates a new line like above.

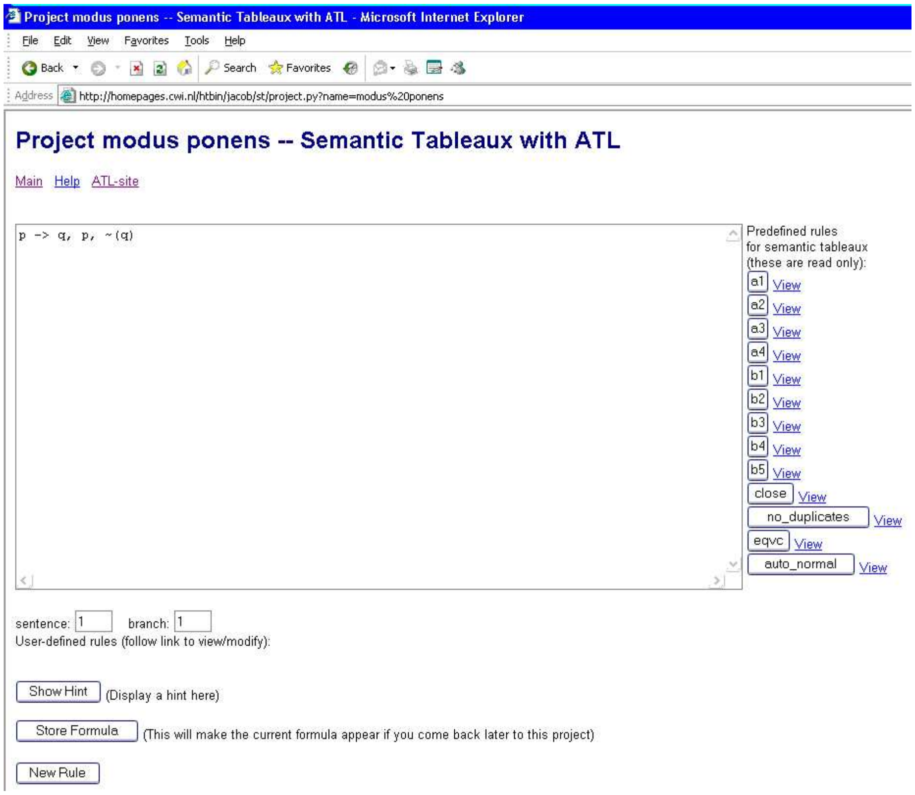
Once more, it is not necessary to implement α– or β–rules yourself, they are predefined for every project since the tool is make for a tableau method.

As an example let's input modus ponens in the application, see if it handles correctly

```
p -> q, p, ~(q)
```

as is done in example project called ”modus ponens” on the website. If you go to this project then you see this formula in the textarea as shown in Figure 7.1.

Pressing the Hint button will make the application try all the defined rules for that project, and suggest one that changes the contents of the textarea. This suggestion is shown beside the Hint button after clicking on it. The application finds the β2–rule as expected, and pressing the b2 button that implements it results in



Project modus ponens -- Semantic Tableaux with ATL - Microsoft Internet Explorer

Address http://homepages.cwi.nl/htbin/jacob/st/project.py?name=modus%20ponens

Project modus ponens -- Semantic Tableaux with ATL

```
p -> q, p, ~(q)
```

Predefined rules for semantic tableaux (these are read only):

sentence: 1 branch: 1

User-defined rules (follow link to view/modify):

(Display a hint here)

(This will make the current formula appear if you come back later to this project)

Figure 7.1: A project window

```
~(p),p,~(q)
q,p,~(q)
```

in the textarea. We see how the $\beta 2$–rule was applied to the first sentence and how the second and third sentences were copied to the new branches. Pressing Hint again will suggest "Close" for the first branch (line), since it contains the contradiction *notp*, written as `~(p)`, and p. Two clicks on the Close button will result in the line "`The proof succeeds.`".

The tool supplies input boxes to choose the branchnumber and the sentencenumber that you wish to apply a rule to. Sometimes only a branchenumber is enough, suppose we want to close the second branch in the example above, we must supply the number 2 in the input box labeled with `branch:`.

Besides the rule buttons, the "Hint" button and a button to add "New Rules" there is a button called "no duplicates" that removes duplicate sentences from a branch, e.g., `p,q,p` will become `p,q`. An explanation of a button is available via the "View" link beside it, in the case of an ATL rule the link leads to the ATL definition. The buttons "eqvc" and "auto normal" are explained in the rest of this Section.

7.3.1 Equivalence classes and conflict relations

We have extended the basic tableau with propositional logic as presented so far, with identity and equivalence classes, in order to add the power of unification. If a branch contains a sentence that is an identity, notation `p = q`, then the user can use the "eqvc" button on that branch to turn this identity into an equivalence class. The notation for an equivalence class is `[p;q]`, with `p` being the canonical representative of the class. Equivalence classes are added upon creation to the end of the line that represents a branch and they can contain more than two instances. If there is already a canonical representative for one of the terms in the identity, then only the new term will be added to the equivalence class. The negation of an equivalence class, notation `~[p;q]` can only contain two terms, it is a conflict relation, and it is also generated via the "eqvc" button, replacing a `~(p = q)` sentence.

With the "auto normal" button every occurence of `q` is rewritten as `p` if `p` is a canonical representative of `q`. While not always strictly necessary for a proof, this can help considerable in makeing a proof better readable, since long formulas can be represented with a single short symbol.

Equivalence classes and conflict relations are useful for unification: a contradictory situation results, after appropriate use of the "eqvc" and "auto normal" buttons, in a conflict–relation of the form `~[P;P]` and that means the branch can be closed.

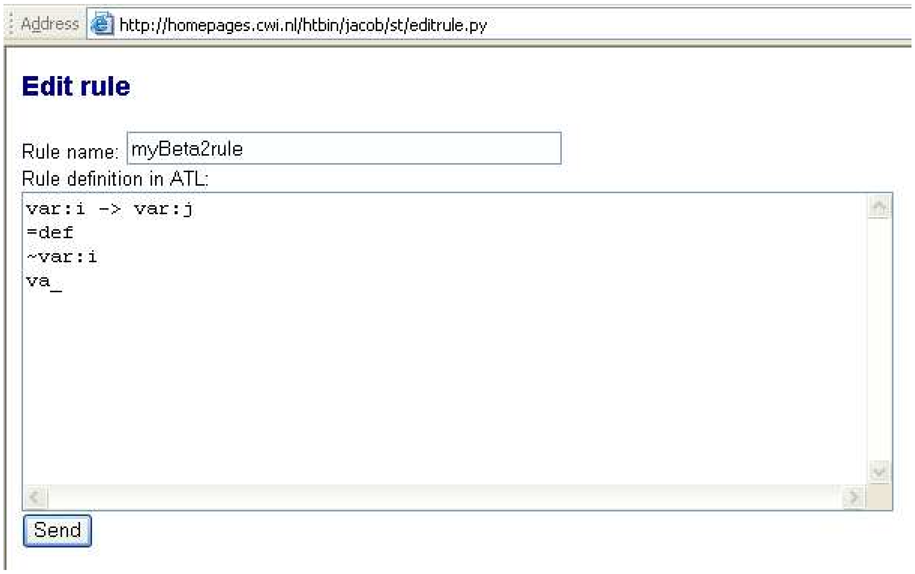
Here are some examples of using the "eqvc" button:

```
p, x = y                         --->    p, [x;y]

p, z = x, [x;y]                  --->    p, [x;y;z]

p, [x;i], [y;j], ~(x = y)        --->    p, [x;i], [y;j], ~[x;y]

p, [x;i], [y;j], ~(i = j)        --->    p, [x;i], [y;j], ~[x;y]
```

As can be seen, a created conflict relation uses the canonical representatives where possible: in the last line `~[i;j]` is not created but `~[x;y]`, since `i` is in an equivalence class with `x` as representative, and likewise for `j` and `y`.

7.3.2 Adding user–defined rules

So far we have only seen formulas in propositional logic and equivalences. In the next section we will see a proof of a property of software and for that proof we wish to use additional notation for the concepts that occur in the proof. New axioms and theories that are needed for the proof must be definable in that new notation, and that is possible with ATL by defining them as rules.



http://homepages.cwi.nl/htbin/jacob/st/editrule.py

Edit rule

Rule name: myBeta2rule

Rule definition in ATL:

```
var:i -> var:j
=def
~var:i
va_
```

Figure 7.2: Editing a rule

The web–application has a ”New Rule” button that leads to a form like in Figure 7.2 where the user can input a new rule and give it a name. After rule submission with the ”Send” button, the new rule appears as a new button with the rulename in the project window under the label ”User defined rules”.

In the next section we show how such new rules can be applied to a derivation. It is the responsibility of the user that a new rule “makes sense”, i.e., has a correct semantics in the context of the notation used. This is more similar to modifying formulas in a proof on paper than to many proof tools where it is not immediately possible to add rules, but only to add axioms. Also, there is no such thing as typechecking a rule in ATL, since we only deal with symbolic computations. For the kind of proofs we encountered, the resulting simplicity was desirable, but your mileage may vary.

In a proof we also want to be able to instantiate axioms, they have to be instantiated with canonical representatives of the equivalence classes in the branch where the axiom is to be placed. For this purpose in the web–application we use the string `axiom` in the antecedent of a rule, the web–application then generates a new button on the project screen just like a new rule button, but now with drop down list input boxes where the user can choose how the axiom is instantiated. In the next Section there are two examples of axioms defined with ATL.

7.4 The Sieve example

In the OMEGA project[OME] we did prove several properties of software. During this work we felt the need for an easy to use tool that can apply a tableau method with equivalence classes and that resulted in the web–application as introduced. As an example we will prove the ”Main sieve property” of software that models the Sieve of Erastosthenes. The software is modeled with UML classes and statemachines and was chosen as an example because it has several interesting characteristics like the dynamic creation of objects. In our proof we abstract from some implementation details and we use a new notation for properties of the software and theories and axioms about those properties.

The notation used is:

`p`	`a sequence of data, a pipeline of sieve objects The complete list of sieve objects or any consecutive part of it (at least 2 elements).`
`sieve(p)`	`The sequence of data resulting from the sieve process applied to p.`
`f(p)`	`The first element of p`
`t(p)`	`The tail of p`
`o`	`A sieve object`
`??(o)`	`The sequence received by object o`
`?(o)`	`sent to o`
`!(o)`	`sent by o`
`<=`	`subsequence operator (infix)`
`~(...)`	`negation of ...`
`[a;...]`	`equivalence class with representative a`
`~[a;b]`	`conflict relation between representatives a and b`

The "Main Sieve Property" can now be written as

```
!(f(t(p))) <= sieve(!(f(p)))
```

meaning that what is sent by a second sieve object in a pipeline, is a subsequence of the sieve process applied to that what is sent by the first sieve object in the pipeline. For this proof we created a project with the name `sieve`.

The proof can be derived automatically by following the "Hint" button as suggested by the tool at each step, except for the input in Step 1, the definitions in Step 2 and the axiom instantiations in Steps 5 and 7.

Besides the functionality provided by the "eqvc" and "auto normal" buttons in the tool, there are four new user defined rules necessary for the proof:

The sieve monotonicity rule:

```
??(var:n) <= ?(var:n)
=def
sieve(??(var:n)) <= sieve(?(var:n))
```

The subsequence transitivity rule:

```
var:p <= var:q,var:q <= var:s
=def
var:p <= var:s
```

Two rules are axioms, they will be instantiated with canonical representatives.

The fifo axiom

```
axiom
=def
??(var:f) <= ?(var:f)
```

The sieve IO axiom

```
axiom
=def
!(var:sio) <= sieve(??(var:sio))
```

The two axioms are to be instantiated with sieve objects in a pipeline as defined.

The proof itself[10]:

[10] right–justified indentation is used in Step 7 to fit the line on the page

```
    Step 1: Take the negation for the semantic tableau method
~(!(f(t(p))) <= sieve(!(f(p))))

    Step 2: Add definitions
~(!(f(t(p))) <= sieve(!(f(p)))), o = f(t(p)), k = f(p), ?(f(t(p))) = !(f(p))

    Step 3: (Hint) eqvc
~(!(f(t(p))) <= sieve(!(f(p)))), [o;f(t(p))], [k;f(p)], [?(f(t(p)));!(f(p))]

    Step 4: (Hint) auto normal
~(!(o) <= sieve(?(o))), [o;f(t(p))], [k;f(p)], [?(o);!(k)]

    Step 5: fifo AXIOM var:f=o
??(o) <= ?(o),~(!(o) <= sieve(?(o))), [o;f(t(p))], [k;f(p)], [?(o);!(k)]

    Step 6: (Hint) RULE sieve monotonicity
sieve(??(o)) <= sieve(?(o)), ~(!(o) <= sieve(?(o))),  [o;f(t(p))], [k;f(p)], [?(o);!(k)]

    Step 7: Sieve IO AXIOM var:sio=o
!(o) <= sieve(??(o)), sieve(??(o)) <= sieve(?(o)), ~(!(o) <= sieve(?(o))),
                                                      [o;f(t(p))], [k;f(p)], [?(o);!(k)]

    Step 8: (Hint) RULE subsequence transitivity
!(o) <= sieve(?(o)), ~(!(o) <= sieve(?(o))), [o;f(t(p))], [k;f(p)], [?(o);!(k)]
    Step 9: (Hint) close: The proof succeeds.
```

7.5 Related work and the future

The Main Sieve Property was also proven with PVS [ORR$^+$96] in the OMEGA project, but that was much more work and resulted in a much longer proof that could not be summarized in a short readable form. The PVS theories incorporated much more detail about the software, and this was necessary to be able to use PVS, so it is unfair to say that the PVS proof was longer or worse. Although about the same topic, the PVS proof was very different and targeted at different goals. Our ATL approach is hard to compare with work in literature, were we find mostly full–blown theorem provers, among many others there are PVS, ACL2 [KMM00] and Isabelle [Pau94]: they are much more powerful and sophisticated. But the sophistication leads to some problems with using those tools in practice: although for instance Isabelle is aiming for human–readable proofs, their success in this respect is questionable, and the, excellently written, ACL2 documentation for example is honest enough to state that it would take months for a highly educated person, familiar with LISP, to use their tool efficiently. The tableau approach with ATL is more like writing a proof on paper, but with the computer helping with the bookkeeping, checking for various types of errors, and providing repetitious tasks like unification. We could not find literature on applying the tableau method to high–level and human–readable proofs like we like to have for software properties, but the tableau world is not our field of specialization and we are very interested if there are people who can point out relevant work that we overlooked.

With respect to future work we would like to replace much of the functionality that is now implemented in the programming language Python in the web–application with a high–level special purpose scripting language that captures ATL rules by name. Such a language can then also be used to implement proof search stategies.

In the OMEGA project there was also a proof on paper created for the example, and the notation chosen in the ATL proof was based on that. By using ATL however we found that some of the theories and axioms as used on the paper proof were not absolutely necessary, and we also were able to avoid having to use induction. Of course, the rules used in the ATL proof would *formally* be needed to be proven themselves, and there the induction would return. However, the rules are rather obvious, and we consider it a nice example of how the tableau method based on ATL can also be used to make a proof shorter or more elegant.

Acknowledgments This work was made possible by the OMEGA [OME] project, an EU sponsored research initiative, IST–2001–33522 OMEGA. Special thanks go to Frank de Boer at CWI for the enlightening discussions.

Part III

Modeling and Analysing Architectures

Chapter 8

Enterprise Architecture Analysis with XML

Authors: F.S. de Boer, M.M. Bonsangue, J.F. Jacob, A. Stam, L. van der Torre

8.1 Introduction

Architectures as defined in the IEEE 1471-2000 standard [Soc00] typically consists of conceptual models visualized as diagrams. Architectural description languages such as UML have been used for information architectures, and more recently similar languages are used for enterprise architectures, such as the Zachman's framework [Zac87]. The research question of this paper is how to design tools for analysis of enterprise architectures. We distinguish between static and dynamic analysis, and we use XML technology. Our approach is based on the following logical concepts[dBBJ$^+$04].

Signature for static analysis. The signature of an architecture focuses on the symbolic representation of the structural elements of an architecture and their relationships, abstracting from other architectural aspects like rationale, pragmatics and visualization. It emphasizes a separation of concerns which allows to master the complexity of the architecture. Notably, the signature of an architecture can easily be expressed in XML for storage and communication purposes, and can be integrated as an independent module with other tools including, e.g., graphics for visualization.

Semantic model for dynamic analysis. The formal *semantics* of a symbolic model of an architecture provides a formal basis for the development and application of tools for the *logical analysis* of the dynamics of an architecture. A signature of an architecture basically only specifies the basic concepts by means of which the architecture is described, but an interpretation contains much more detail. In general, there can be a large number of different interpretations for a signature. This reflects the intuition that there are many possible architectures that fit a specific architectural description.

By applying the techniques for static and dynamic analysis discussed in this paper, we get a better understanding of how enterprise architectures are to be interpreted and what we mean with the individual concepts and relationships. In other words, these techniques allow enterprise architects to validate the correctness of their architectures, to reduce the possibility of misinterpretations and even to enrich their architectural descriptions with relevant information in a smooth and controllable way.

The layout of this paper is as follows. In Section 8.2 we introduce a running example to explain our definitions. In Section 8.3 we discuss tool support, XML, AML and RML. In Section 8.4 and 8.5 we explain static and dynamic analysis using these tools.

8.2 ArchiMate: a running example

To illustrate static and dynamic analysis in enterprise architectures, we use an example from the ArchiMate project. ArchiMate is an enterprise architecture modelling language [JvBA$^+$03, ea04]. It provides through a metamodel concepts for architectural design at a very general level, covering for example the business, the application, and the technology architecture of a system. The Archmate language resemble the business language Testbed [EJL$^+$99] but it has also a UML-flavor, introducing concepts like interfaces, services, roles and collaborations.

The example modelled using the ArchiMate language concerns the enterprise architecture of a small company, called *ArchiSell*. In *ArchiSell*, employees sell products to customers. The products are delivered to ArchiSell by various suppliers. Employees of ArchiSell are responsible for ordering products and for selling them. Once products are delivered to ArchiSell, each product is assigned an owner, responsible for selling the product.

To describe this enterprise we use the ArchiMate concepts and their relationships as presented in Figure 8.1. In particular, we use structural concepts (product, role and object) and structural relationships (association), but also a behavioral concepts (process) and behavioral relationships (triggering). Behavioral and structural concepts are connected by means of the assignment and access relationships.

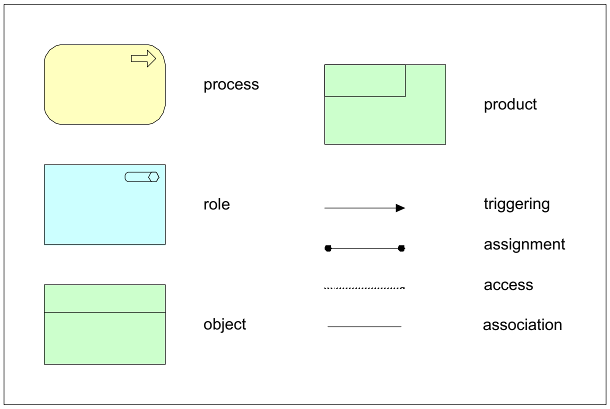


Figure 8.1: Some concepts and relations

A product is a physical entity that can be associated with roles. A role is the representation of a collection of responsibility that may be fulfilled some entity capable of performing behavior. The assignment relation links processes with the roles that perform them. The triggering relation between process describes the temporal relations between them. When executed, a process may need to access data, whose representation is here called object.

We specifically look at the *business process architecture* for ordering products, depicted in Figure 8.2. In order to fulfill the business process for ordering a product, the employee has to perform the following activities:

- Before placing an order, an employee must register the order within the Order Registry.

- After that, the employee places the order with the supplier.

- As soon as the supplier delivers the product(s), the employee first checks if there is an order that refers to this delivery. Then, he/she accepts the product(s).

- Next, the employee registers the acceptance of the product(s) within the Product Registry and determines which employee will be the owner of the product(s).

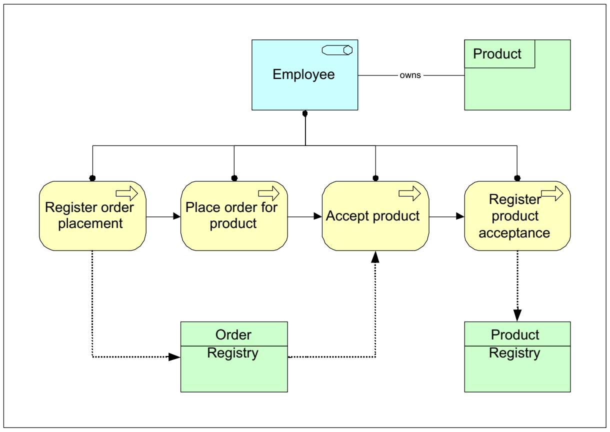


Figure 8.2: A Business Process Architecture

Despite the apparent simplicity of the diagram, there are several issues which can be analyzed. For example, when an architect presents this architecture, he may explain that the role of the order registry is to coordinate between the first two processes of placing orders and accepting them. Whereas the same employee should see to it that an order which is placed is also registered, there may be another employee which accepts the order.

Also variants can be analyzed. For example, given the fact that the coordination between order placement and order acceptance is regulated via the order registry, is it still necessary that placing the order for a product triggers the process that accepts the product. In other words, what is the impact if we change the architecture by removing this relation?

Before we can consider these questions, we need a language to represent the architecture. The ArchiMate language is a visual modelling language not well suited for representation or reasoning. We therefore represent architectures like the one above in XML.

8.3 The tools: XML, AML and RML

Before we start to analyze the enterprise architecture of the running example, we introduce our machinery. It consists of XML, AML and, most importantly, RML.

The Extensible Markup Language (XML) [XML] is a universal format for documents containing structured information using nested begin and end labels, which can contain attributes. For example, a such as:

```
<product>
<weefer color="green">zyx</weefer>
<wafer color="blue">cis</wafer>
<weefer color="green">zyx</weefer>
</product>
```

The nested structure of the labels corresponds to a tree. They can be used over the internet for web site content and several kinds of web services. It allows developers to easily describe and deliver rich, structured data from any application in a standard, consistent way. Today, XML can be considered as a lingua franca in computer industry, increasing interoperability and extensibility of several applications. Terseness and human-understandability of XML documents is of minimal importance, since XML documents are mostly created by applications for importing or exporting data.

The ASCII Markup Language (AML) [Jaca] used to show examples in this paper is an alternative for XML syntax. AML is designed to be concise and elegant and easy to use. AML uses indentation to increase readability and to define the XML tree hierarchy: indentation level corresponds to depth, sometimes called level, in the tree. No indentation is required for the set of attributes that immediately follows each attribute name.

```
product
  weefer color="green"
    zyx
  wafer color="blue"
    cis
  weefer color="green"
    zyx
```

The Rule Markup Language (RML) is a tool for transforming XML documents that can be used for analysis of architectural description, and in particular for the definition and simulation of the system behavior. It consists of a set of XML constructs that can be added to an existing XML vocabulary in order to define RML rules for that XML vocabulary. These rules can then be executed by RML tools to transform the input XML according to the rule definition. Consider for example the following rule which removes duplicates from an XML document.

```
div class=rule name="Removeduplicates"
    div class=antecedent
        product
            rml-list name=rml-A
            rml-tree name=rml-B
            rml-list name=rml-C
            rml-tree name=rml-B
            rml-list name=rml-D
    div class=consequence
        product
            rml-use name=rml-A
            rml-use name=rml-B
            rml-use name=rml-C
            rml-use name=rml-D
```

The example illustrates the main constructs. First, there is an input and an output part of the rule, called antecedent and consequent. The antecedent contains a set of variables, rml-A, rml-B, rml-C and rml-D. The second variable occurs twice, and will therefore only match with a duplicate. Finally, rml-list matches with a list of elements, and rml-tree with one element; they can be considered the analogues of * and ? in regular expressions as used in for example grep.

The antecedent matches with the product given above, and binds the variables such that rml-A and rml-E are empty, rml-B is the seefer and rml-C is the woofer. The consequent of the rule explains the output of the rule. It reproduces the content otf the variables rml-A, rml-B, rml-C and rml-D, but it does not reproduce the second instance of rml-B. In this way, the duplicate is removed.

```
product
  weefer color="green"
    zyx
  wafer color="blue"
    cis
```

There are a few more constructs, dealing for example with variables for attributes such as color. The set of RML constructs is concise and shown in

Table 2.1. Things that can be stored in RML variables are element names, attribute names, attribute values, whole elements (including the children), and lists of elements.

The example illustrates that a pattern can be matched that is distributed over various parts of the input XML. Such pattern matching is hard to define with other existing approaches to XML transformation because they do not use of the problem domain XML for defining transformation rules: transformations are defined either in special purpose language like the Extensible Stylesheet Language Transformation (XSLT), or they are defined at a lower level by means of programming languages like DOM and SAX. RML captures transformations defined by a single rule, but interaction among rules is dealt with by other tools. Moreover, XML transformations normally involve creating links between elements by means of cross-referencing attributes, or reordering elements, or adding or removing elements, but does typically not include things like integer arithmetic and floating point calculations. In case of such transformations RML tools will have to be combined with other tools that can do the desired calculation.

Combinations of RML with other components like programming language interpreters has been applied successfully in the EU project OMEGA (IST-2001-33522, URL: http://www-omega.imag.fr) that deals with the formal verification of UML models for software. That tool for the simulation of UML models does the XML transformations with RML, and uses an external interpreter for example for floating point calculations on attributes in the XML encoding.

In the remainder of this paper, we show how RML can be used for the analysis of the enterprise architecture in the running example. RML was designed to make the definition of executable XML transformations also possible for other stakeholders than programmers. This is of particular relevance when transformations capture for instance business rules. In this way it is possible to extend the original model in the problem domain XML vocabulary with semantics for that language. Similarly, it is also possible to define rules for constraining the models with RML.

As illustrated above, with RML a formal definition can be given of the dynamics of the basic actions of an architecture in terms XML transformations.

The most widely used mathematical foundation for describing semantics is the Structural Operational Semantics by Plotkin [Plo81], and this is what we use for the specification of the behavior of an architecture in XML in

Sect. 8.5.

Below we show an example of RML by presenting the rule that defines the state transformation of the action of our running example, where emp and order-reg are individual names for an employee and the Order_registry, respectively. The details of this notation are discussed later in this paper.

```
emp, order-reg := Register_order_placement(
        emp, order-reg)
```

Content-preserving RML constructs have been omitted for clarity.

```
div class=rule name="Register order placement"
    div class=antecedent
        variables
            rml-Employee order=rml-OrderName
                product=rml-ProductName
            order-registry
                rml-list name=oldOrders
    div class=consequence
        variables
            rml-Employee order=rml-OrderName
                product=rml-ProductName
            order-registry
                rml-use name=oldOrders
                order name=rml-OrderName
```

This example illustrates several RML constructs which do not appear in the removal of duplicates example. In particular, it uses variables for element names and atribute values. The effect of applying this rule is that order-registry is extended with an order.

In the `antecedent` of the rule the matching algorithm first looks for an element with name `variables` which contains that part of the AML representation of the semantic model that stores the values of the names emp (of sort Employee) and order-reg (of sort Register_order). For instance, a value of emp is an XML element like

```
e1 order=Product product=p1
```

where `Product` is a sort in the architecture and `p1` is an individual product.

If the `variables` element is found the matching algorithm looks for children of that element: one child with an `order` and `product` attribute (an employee), and one child with the name `or` (the order registry). The algorithm binds the employee name emp to RML variable `Employee` and it binds the values of the `order` and `product` attributes to `OrderName` and `ProductName` respectively. The list of old orders, a list of XML elements that are the children of the `orders` child of the `r1` order registry, is bound to RML variable `oldOrders`. In the consequence of the rule the variables are reused in the output and an order element with the correct name is appended to the `oldOrders` list. Note that by means of this RML rule we have an interpretation of the sort Order_registry of *unbounded* capacity.

Section 8.2 described a model of an architecture in a typical bussiness-like fashion: with diagrams and English text for additional explanations. In Sections 8.4 and 8.5 we will show how we can use XML for this description, resulting in a *formalization* of the model. There are *static* and *dynamic* aspects to the model: the static aspects give the structure of the model, the dynamic aspects describe how the model can change.

8.4 Static analysis

We designed our own XML vocabulary, because we could not find an adequate standard one. We base this design on a formal basis discussed in Sect. 8.4.1. Diagrams like the one in Fig. 8.2 can be viewed in an abstract way as consisting of nodes and arrows, where some of the arrows are bidirectional. In the architectural community the nodes are called concepts and the arrows are called relations. Depending on the topic of the diagram, in some cases there is an existing *standardized* XML vocabulary that can be used to provide an XML encoding of the diagram, for instance there is XMI to encode UML diagrams. What is typically lost in such an encoding are some of the visual elements: the positions of the boxes in the picture and the lengths of the lines for the arrows. An XML encoding only captures the names of the nodes and the arrows and what nodes are connected via which arrows. There can also be information in the XML encoding about attributes of the nodes and arrows, information that is not visible in the diagram but in the accompanying text in English. An example of such extra information is that a department consists of a maximum of 100 employees.

8.4.1 A formal basis for static analysis

Following IEEE 1471-2000 [Soc00], every system has an architecture. In our perspective which abstracts from pragmatics, like design principles, an architecture is the structure and dynamics of a system consisting of its components and their relationships.

The architecture of a system is purely conceptual and different from particular symbolic descriptions of that architecture. An architectural description consists of several symbolic models (also called model in [Soc00]) and other pragmatic information. Examples of the latter are the architectural rationale. In the next sections we focus on the logical nature of these symbolic models which involves their syntax and semantics.

The core of a symbolic model of an architecture consists of its *signature* which specifies its *name space*. The names of a signature are used to denote symbolically the structural elements of the architecture, their relationships, and their dynamics. The *nature* of each structural element is specified by a *sort*, and each architectural relationship by a *relation* between sorts. Additionally, a signature includes an ordering on its sorts and its relations for the specification of a classification in terms of a generalization relation on the structural elements and the architectural relations. For example, the sort object in Figure 8.1 can be defined as a generalization of both the sorts Order_Registry and Product_Registry given in Figure 8.2, to indicate that every element in Order_Registry or Product_Registry is also an element of sort object. Also, an association between role and product is a generalization of the relation owns between Employee and Product.

The ordering on sorts and relations is in general used to capture certain aspects of the *ontology* of an architecture. Other ontological aspects can be captured by the aggregation and containment relations. For technical convenience however we restrict to the generalization relation only.

Definition 4 *A signature consists of*

- *a partially ordered set of* primitive sorts*, also called the* sort hierarchy*;*
- *a partially ordered set of* relations*, where each relation is of the form $R(S_1, \ldots, S_n)$, with R the name of the n-ary relation and S_i the primitive sort of its ith argument.*

We allow *overloading* of relation names, i.e., the same name can be used for different relations. For instance, given the primitive sorts

$Person$, $Boss$, and $Employee$, the relations $Responsible(Boss, Employee)$ and $Responsible(Person, Person)$ are in general two different relations with the same name.

Further information about the architecture is expressed symbolically in terms of suitable extensions of one of its signatures. Usually a signature is extended with operations for constructing complex *types* from the primitive sorts. Examples are the standard type operations like *product type* $T_1 \times T_2$ of the types T_1 and T_2, and the *function type* $T_1 \rightarrow T_2$ of all functions which require an argument of type T_1 and provide a result of type T_2. Note that a relation $R(S_1, \ldots, S_n)$ is a sub-type of $S_1 \times \cdots \times S_n$.

Given functional types, the name space of a signature can be extended with *functions* $F(T_1) : T_2$, where F specifies the name of a function of type $T_1 \rightarrow T_2$. Functions can be used to specify the *attributes* of a sort. For example, given the primitive sorts $Employee$ and $\mathbb{N}$, the function $Age(Employee) : \mathbb{N}$ is intended for specifying the age of each person. Note that *multi-valued* functions $F(T_1, \ldots, T_n) : T'_1, \ldots, T'_m$ can be specified by the functional type $T \rightarrow T'$, where T denotes the product type $T_1 \times \cdots \times T_n$ and T' denotes the product type $T_1 \times \cdots \times T_n$. In general, functions are also used to specify symbolically the dynamics of an architecture.

The next example shows the signature of the business process architecture described in Figure 8.2.

Example 1 *The sorts of the example described in Figure 8.2 and 8.1 are simply enumerated by*

```
process
role
object
product
Employee
Product
Order_Registry
Product_Registry
```

Note that we did not include processes as a sort. The subsort relation is specified in AML by the following enumeration

```
is-a
    domain name=Employee
    codomain name=Role
```

```
is-a
    domain name=Order_Registry
    codomain name=Object
is-a
    domain name=Product_Registry
    codomain name=Object
is-a
    domain name=owns
    codomain name=assignment
```

Note that we have encoded meta-model information of an architecture as part of the signature of the architecture itself. The relation between the meta-model sorts and relations and architectural sorts and relations is expressed by the respective partial orders between sorts and relations of the signature.

In AML the owns-relation itself is specified by

```
owns
    domain name=Employee
    codomain name=Product
```

Finally, the processes are specified in AML as functions. The types of the arguments and result values are determined as follows: A role which is assigned to a process specifies the type of both an argument and a result value of the corresponding function. Similarly, an outgoing access relation from a process to an object specifies the type of both an argument and a result value of the corresponding function. On the other hand, an incoming access relation from an object to a process only specifies the type of the corresponding argument (this captures the property of 'read-only').

```
Register_order_placement
    domain name=Employee
    domain name=Order_Registry
    codomain name=Employee
    codomain name=Order_Registry
Place_order_for_product
    domain name=Employee
    codomain name=Employee
Accept_product
    domain name=Employee
    domain name=Order_Registry
    codomain name=Employee
Register_product_acceptance
    domain name=Employee
domain name=Product_Registry
codomain name=Employee
codomain name=Product_Registry
```

Note that the triggering relation is not included in our concept of a signature. In our view such a relation specifies a temporal ordering between the processes which is part of the business process language discussed below in section 8.5.

Interpretation of types We first define a formal interpretation of the types underlying a symbolic model.

Definition 5 *An* interpretation I *of the types of a signature assigns to each primitive sort* S *a set* $I(S)$ *of* individuals *of sort* S *which respects the subsort ordering: if* S_1 *is a subsort of* S_2 *then* $I(S_1)$ *is a* subset *of* $I(S_2)$.

Any primitive sort is interpreted by a subset of a universe which is given by the union of the interpretation of all primitive sorts. The hierarchy between primitive sorts is expressed by the subset relation.

An interpretation I of the primitive sorts of a signature of an architecture can be inductively extended to an interpretation of more complex types. For example, an interpretation of the product type $T_1 \times T_2$ is given by the Cartesian product $I(T_1) \times I(T_2)$ of the sets $I(T_1)$ and $I(T_2)$. The interpretation of the function type $T_1 \rightarrow T_2$ as the set $I(T_1) \rightarrow I(T_2)$ of all functions from $I(T)1)$ to $I(T_2)$, however, does not take into account the *contra-variant* nature of the function space. For example, since the sort $\mathbb{N}$ of natural numbers is a sub-sort of the real numbers $\mathbb{R}$, a function from $\mathbb{R}$ to $\mathbb{R}$ dividing a real number by 2 is also a function from $\mathbb{N}$ to $\mathbb{R}$, but, clearly, the set of all functions from $I(\mathbb{R})$ to $I(\mathbb{R})$ is *not* a subset of the set of functions from $I(\mathbb{N})$ to $I(\mathbb{R})$.

Therefore, given the universe $\mathbb{U}$ defined as the union of all the interpretations of the primitive sorts, we define the interpretation of the function type $T_1 \rightarrow T_2$ by

$$I(T_1 \rightarrow T_2) = \{f \in \mathbb{U} \rightarrow \mathbb{U} \mid f(I(T_1)) \subseteq I(T_2)\}.$$

The function type $T_1 \rightarrow T_2$ thus denotes the set of all functions from the universe to itself such that the image of $I(T_1)$ is contained in $I(T_2)$. Note that if T_1' is a subtype of T_1 and T_2 is a subtype of T_2' then $I(T_1 \to T_2)$ is indeed a subset of $I(T_1' \to T_2')$.

In general, there can be a large number of different interpretations for a signature. This reflects the intuition that there are many possible architectures that fit a specific architectural description. In fact, a signature of an architecture basically only specifies the basic concepts by means of which the architecture is described.

Semantic models In our logical perspective, a semantic model is a formal *abstraction* of the architecture of a system. The logical perspective presented until now, only concerned the symbolic representation of an architecture by means of its signature. Next we show how to obtain a formal model of a system as a semantic interpretation of the symbolic model of its architectural description.

The semantic model of a system involves its concrete components and their concrete relationships which may change in time because of the dynamic behavior of a system. To refer to the concrete situation of a system we have to extend its signature with names for referring to the individuals of the types and relations. For a symbolic model, we denote by $n : T$ a name n which ranges over individuals of type T.

Given a symbolic model of an architecture extended with individual names and an interpretation I of its types, we define a semantic model Σ as a function which provides the following interpretation of the name space of the symbolic model covering its relations, functions, and individuals.

Relations For each relation $R(S_1, \ldots, S_n)$ we have a relation

$$\Sigma(R) \subseteq I(S_1 \times \cdots \times S_n)$$

respecting the ordering between relations, meaning that if R_1 is a subrelation of R_2 then $\Sigma(R_1)$ is a subset of $\Sigma(R_2)$.

Functions For each symbolic function $F(T_1) : T_2$ we have a function

$$\Sigma(F) \in I(T_1 \to T_2).$$

Variables For each individual name $n : S$ we have an element

$$\Sigma(n) \in I(S).$$

8.4.2 XML for static analysis

In this section we describe the methodology we follow to design an XML vocabulary for diagrams like in Fig. 8.2 and 8.1. In general we will model every node in the diagram with an XML element. Figure 8.1 is a legenda, a collection of unconnected concepts and relation names with their visual representation. Only the concepts are given XML elements, not the relation names. For the concepts (rectangles and rounded rectangles) in Fig. 8.1 and 8.1 we design XML elements with that name. The lines in Fig. 8.2, and other relations that are mentioned in the accompanying text, will be modeled with XML elements with the name of the relation, and these elements will have `domain` and `codomain` children that contain cross-references to the elements that participate in the relation. This way it is possible to define n to m relations by taking n `domain` elements and m `codomain` elements. A designer could choose to take other names for `domain` and `codomain`, like `from` and `to`, but the methodology remains the same.

Section 8.4.1 shows examples for the various XML elements in the model. The complete XML model for static analysis for the example consists of a `businessprocess` element with as children elements the examples in Sect. 8.4.1.

All the concepts and relations from Fig. 8.2 and 8.1 and the explanatory text have been put into XML. The disadvantage of storing meta-information in an XML encoding, like in this case with `is-a` relations, is that the encoding risks to become too big and chaotic. The chaos can be improved upon with extra elements, for instance by putting the meta concepts (`process`, `role`, `object` and `product`) in a containing element called `meta`, but this still does not solve the size problem. If analysis is not using the meta information, then it can be omitted, or stored in an external file for future reference. In the above model this method would remove all the `is-a` relations and the four meta elements.

Our XML encoding does not make much use of the possibilities to use hierarchy between elements in XML itself. An example of using more XML hierarchy would be:

```
businessprocess
    role
        Employee
    object
        Order_Registry
        Product_Registry
    product
        Product
    process
        Register_order_placement
            domain name=Employee
            domain name=Order_Registry
            codomain name=Employee
            codomain name=Order_Registry
        Place_order_for_product
            domain name=Employee
            codomain name=Employee
        Accept_product
            domain name=Employee
            domain name=Order_Registry
            codomain name=Employee
        Register_product_acceptance
            domain name=Employee
            domain name=Product_Registry
            codomain name=Employee
            codomain name=Product_Registry
    owns
        domain name=Employee
        codomain name=Product
```

which is a more efficient encoding for the example, but our experience shows that it is generally a good idea to be cautious when using XML hierarchy. With this last encoding it will be more difficult for example to put the meta information in a separate file. And there are several kinds of relations in a model, like generalization, composition and association, that can be expressed with hierarchy in XML, but once we have chosen to use hierarchy in XML for generalization it will not readily be possible to use XML hierarchy also for composition relations when we want to add those later. In the case of modeling generalization there is also the problem of modeling what is known as “multiple-inheritance” in computer science: it is not generally possible to model a generalization of two concepts with XML hierarchy alone because an XML element only has one parent element. If generalization is very important and interesting for the analysis you have in mind then modeling it with XML hierarchy could possibly work out very well, but in our methodology we start out using as little XML hierarchy as possible.

XML individuals for semantic models So far we have only put sorts and relations into XML, but not individuals of sorts, necessary for semantic models. Putting the individuals into XML can be useful for several types of analysis, especially for analysing dynamics. In our methodology we can model individuals of a sort as XML children of the sort element, with all attributes that are needed as can be inferred from the text description of an architecture. The name of the children element is free to choose, but there could be a naming convention such that it is clear what sort an individual belongs to. For example, adding two individuals of sort `Employee` can be modeled with:

```
businessprocess
    ...
    Employee
        e1 order=Product product=p1
        e2 order=Product product=p2
    ...
```

where the `e1` and `e2` elements are `Employee` individuals and their `order` and `product` attributes have been added because the textual description of the architecture said that an employee has an order in mind and that an employee is handling a product. There is only one `Product` sort in our example, so the `order` attribute looks redundant, but we may want to add more products later.

Another approach is to put all the XML elements for sort individuals inside a `variables` element, and in that case it would be a good idea to give the individuals an attribute that designates their sort, like in

```
businessprocess
    ...
    Employee
    ...
    variables
        e1 sort=Employee order=Product product=p1
        e2 sort=Employee order=Product product=p2
        ...
```

where we see the use of an extra `sort` attribute. Of course another name than `variables` is possible. And of course their are many different approaches altogether, but with the two described here we have good experiences.

Examples of static analysis An example of static analysis is to analyse whether all name attributes of domain and codomain elements in the functions are defined as XML element names, and to do type checking if that is considered useful. Another example is to check if all the `is-a` relations are anti-symmetric. Yet another example is *impact analysis*.

To perform the static analysis there are many tools in the industry that can be used that are capable of parsing XML. These tools can be used to turn the XML in a graphical representation, or they can do things like counting the number of employees or adding their salary attributes. The RML tools can also be used. The RML tools are designed for *transformations* of XML to XML so they are more targeted at dynamic analysis, but it is very well possible to define transformations of XML that rearrange the input: for example displaying a list of employee elements. Due to a lack of space we can not already show examples of such RML transformations here, we refer to Sect. 8.5.2 for RML examples.

8.5 Dynamic analysis

8.5.1 A formal basis for dynamic analysis

We can model the dynamic behavior of a model of an architecture with a state-machine [GBR99]. The transitions in the state-machine correspond with RML rules or recipes.

State machine semantics The sort individuals are coordinated by means of state machines. These state machines consist of *transitions* of the form

$$l \xrightarrow{[g]/a} l'$$

where l is the *entry* location and l' is the *exit* location of the transition. Furthermore, g denotes its boolean *guard* and a its *action*.

The boolean guard of a transition is a boolean expression that consists of the usual integer values and string values but also of RML-variables from the rule or recipe that is captured by the transition. For evaluating the guard these RML-variables will be assigned a value by the RML matching algorithm with the XML encoding of the model as input.

An action involves a call to the RML tools executing an RML rule or recipe on the model. For the action in the transition we generally use the name of the file the rule or recipe is stored in.

In the following we use *class* for sort and we use *object* for individual, because these names are more usual when describing state-machines, e.g. in UML.

In order to formally define the *operational* semantics of state machines in architectures we assume for each class c of a given architecture a set O_c of *references* to objects in class c. In XML such references can be modeled by means of `id` attributes with unique values, and cross-reference attributes. In case class c extends c' (according to the architecture) we have that O_c is a subset of $O_{c'}$. (For classes which are not related by the inheritance hierarchy these sets are assumed to be disjoint.)

Definition 6 *An* object diagram *of a given architecture with classes $c_1, \ldots, c_n$ can be specified mathematically by functions σ_c, for $c \in \{c_1, \ldots, c_n\}$, which specify for each object in class c existing in the object diagram the values of its attributes, i.e., $\sigma_c(o.A)$ denotes the value of attribute A of the object o, i.e., it denotes an object reference in $O_{c'}$, where c' is the (static) type of the attribute A (defined in the class c in the architecture).*

Often we omit the information about the class and write simply $\sigma(o.A)$. Control information of each object o in an object-diagram is given by $\sigma(o.L)$, assuming for each class an attribute L which is used to refer to the current location of the state machine of o.

Given an architecture consisting of a finite set of classes $c_1, \ldots, c_n$ and a state machine, we define its behavior in terms of a *transition relation* on the object diagram.

This transition relation is defined parametric in the semantics of the application operations.

More specifically, we assume for each action a involving an RML rule or recipe a *labeled* transition relation $\sigma \xrightarrow{a} \sigma'$ which specifies σ' as a possible result of the execution of the call a on σ.

Such a labeled transition describes the *observable* effect on the architecture of the execution of the corresponding call by the RML tools. As a special case we assume for each *guard* g a *labeled* transition relation $\sigma \xrightarrow{g} b$

where b denotes a boolean value which indicates the result of the evaluation.

Definition 7 *Formally, given an architecture and the semantic interpretations of the RML rules and recipes, we have a transition $\sigma \rightarrow \sigma'$ from the object-diagram σ to the object-diagram σ' if the following holds: there exists an object o and a transition*

$$l \xrightarrow{[g]/a} l'$$

in its state machine such that

Location $\sigma(o.L) = l$ *and* $\sigma'(o.L) = l'$*;*

Guard $\sigma \xrightarrow{g}$ *true;*

Action *in case of a call a involving an RML rule or recipe we have*

$$\sigma \xrightarrow{a} \sigma'.$$

The first clause above describes the flow of control. The second clause states that the guard evaluates to true (without side-effects). A call to an RML rule or recipe is described in terms of a corresponding labeled transition which models the execution of the call by the underlying RML tools. Note that the execution of a transition of a state-machine is atomic. However, more fine-grained modes of execution can be introduced in a straightforward manner.

8.5.2 XML+RML for dynamic analysis

In our methodology we start with writing out scenarios. Scenarios consist of sequences of semantic models, called *scenes*, connected by functions, called *transitions*. We use the words scene and transition or transformation when discussing XML encodings. An example is an employee who registers an order in the order registry: the source-scene of the transition contains an employee with an order and an order registry, the target-scene contains the employee and the order registry with the order added. When we have collected enough examples of transitions, we define the RML rules that define the XML transformations from scene to scene. We could also try to define the RML rules without collecting scenes first, but using scenes has proven to be useful in practice and the scenes also provide a testbed to try the rules on, and later versions of rules. From source- and target-scene to an RML rule often does not involve much more than replacing literal strings with RML variables. The resulting set of RML rules can be used as actions in state-machines to define the behavior of an architecture. If a particular transition is too complex for 1 rule then a sequence of possibly iterating rules can be collected in an RML recipe, and the recipe can then be used as the action in the transition of a state-machine.

We now demonstrate our methodology applied to the "Register order placement" process in the running example.

The XML contain a `businessprocess` element as shown before containing the sorts and relations from the symbolic model and a `variables` element where we keep the sort individuals. To save space we only show the `variables` section from now on.

A first scene consists of an employee and an order registry:

```
variables
    e1 sort=Employee order=Product
    order-registry
        Product
        Product
```

The XML element with the name `e1` corresponds to an *emp:Employee* variable in Sect. 8.5.1 and the XML element with the name `order-registry`, with its children, corresponds to a *or:Order_Registry* variable. These variables are parameters of a function *Register_order_placement* like in Sect. 8.5.1.

From this scene, the register order placement process leads to another scene:

```
variables
    e1 sort=Employee order=None
    order-registry
        Product
        Product
        Product
```

where the order attribute Employee is now None and the order for a Product has been added to the registry.

To produce a simplistic RML rule based on only these two scenes, we define

```
div class=rule name="Register order placement"
    div class=antecedent
        variables
            e1 sort=Employee order=Product
            order-registry
                Product
                Product
    div class=consequence
        variables
            e1 sort=Employee order=None
            order-registry
                Product
                Product
                Product
```

as the first version of the RML rule we want to develop for the process. To create this rule we simple copied the first scene in the antecedent of the rule, and we copied the second scene in the consequence.

This RML rule works, but only for employee elements with the name `e1`, and only for products of type `Product` as value of the order attribute of the employee. There could be other products e.g. `Product2` in the symbolic model and such products as value of the order attribute will not work. And the rule would only work when there are exactly 2 Products already in the registry where we want the rule to work with any number in the registry already. We can see these other possibilities by looking at other possible source scenarios we collected around this process.

```
variables
    e1 sort=Employee order=Product
    e2 sort=Employee order=Product2
    order-registry
        Product
        Product


variables
    e2 sort=Employee order=Product2
    order-registry
        Product
```

To make the rule work also on these other scenarios, we change the relevant literal strings in the rule into RML variables, according to table 2.1, leading to the second version of the rule:

```
div class=rule name="Register order placement"
    div class=antecedent
        variables
            rml-Employee sort=Employee
                         order=rml-P
            order-registry
                rml-list name=OldOrders
    div class=consequence
        variables
            rml-Employee sort=Employee
                         order=rml-P
            order-registry
                rml-use name=OldOrders
                Product
```

This rule is much better, but still not finished. This rule only works if there is exactly 1 employee sort individual defined and exactly 1 order-registry. But there could be other things defined in the variables section around the employee elements (we assume that an order-registry is always last in the variables section). If there are, the rule will not work since the first element does not match the pattern for an employee element as defined, or the second element is not an order-registry element. To copy such other elements in the variables section we change the rule,

```
div class=rule name="Register order placement"
    div class=antecedent
        variables
            rml-list name=Pre
            rml-Employee sort=Employee
                         order=rml-P
            rml-list name=Post
            order-registry
                rml-list name=OldOrders
    div class=consequence
        variables
            rml-use name=Pre
            rml-Employee sort=Employee
                         order=rml-P
            rml-use name=Post
            order-registry
                rml-use name=OldOrders
                Product
```

putting everything before the employee we want to match in RML variable Pre and putting everything after it, except the last element that must be order-registry, in Post.

A final addition to the rule is needed because an employee pattern in the rule now has a sort and a order attribute, but could very well have other attributes we want to keep in the output. This is done by adding an attribute rml-others=Others to the rml-Employee elements in the antecedent and in the consequence.

Now that we have defined this rule, we can define the first transition of the state-machine for this business process. To do this in XML we add a statemachine element to the businessprocess element, and with this first transition it looks like:

```
statemachine
    transition id=t1
        source state=start
        target state=state_1
        action
            implementation
                """Register order placement"""
```

When we have modeled the whole running example, there will be 4 transitions in the state machine, for the 4 processes in Fig. 8.1. A transition does not have to consist of an action alone, there can also be a guard with an guard-expression containing the usual things like string values and integers, but also RML variable names from the RML rule in the action. The guard-expression can be for example a Java expression that can be evaluated by a Java interpreter, or it can be an OCL expression, or anything else suitable. The purpose of such a guard-expression is to constrain the applicability of the RML rule. For example to add the constraint that only orders of sort Product2 or Product3 may be added, a guard is added to the t1 transition, resulting in:

```
state machine
    transition id=t1
        source state=start
        target state=state_1
        guard
            implementation
"""P == 'Product2' or P == 'Product3'"""
```

```
    action
        implementation
            """Register order placement"""
```

We can not give more examples here due to a lack of space. An online executable demonstration of an extended version of such a state-machine can be seen at http://homepages.cwi.nl/ jacob/km/cgikm.html, where it is a state-machine for UML models. The only difference with a state-machine we need for business processes is that the action.implementation elements in the UML models contain statements in a programming language, where we only use the name of RML rules in this paper, and the UML state machines are more complex since they also handle events with triggers. For business process modeling we do not need events and triggers since there is only one active process.

8.6 Summary and outlook

The techniques proposed in this paper enforce architects to think about the relation between their architectures and the real world. Static analysis techniques allow them to think about structural issues, like cardinality and "is-a" relationships. With dynamic analysis techniques, they can make small simulations the processes or other behavioural descriptions they propose. All these techniques improve the understanding of their own creations.

In this paper we have introduced a XML tool for static and dynamic analysis of enterprise architectures. We have shown how it transforms XML data and how it can be used to simulate business processes. A summary of the methodology we follow:

1. Create a symbolic model, see Sect. 8.4.2.
2. Collect scenes (semantic models) around transitions (functions).
3. Create RML rules using copy and paste from scenes.
4. Replace strings by RML variables in the RML rules where needed.
5. Create state-machines with the RML rules as actions in the state-machine transitions.

There is a rich literature of business processes. However, as far as we know, our logical perspective is a first attempt to a formal integration of such processes in enterprise architectures. We believe that our logical framework (plus tool support) also provides a promising basis for the further design and development of business process languages and corresponding tools.

Acknowledgements This paper results from the ArchiMate project (`http://archimate.telin.nl`), a research initiative that aims to provide concepts and techniques to support architects in the visualization, and analysis of integrated architectures. The ArchiMate consortium consists of ABN AMRO, Stichting Pensioenfonds ABP, the Dutch Tax and Customs Administration, Ordina, Telematica Institute, CWI, University of Nijmegen, and LIACS.

Chapter 9

A Logical Viewpoint on Architectures

Authors: F.S. de Boer, M.M. Bonsangue, J.F. Jacob, A. Stam, L. van der Torre

9.1 Introduction

In this paper we consider the gap between abstract enterprise architecture descriptions and much more detailed business process models. The problem of analyzing and simulating enterprise architectures is that they are described in much more vague terms than business process models. For example, the IEEE standard 1471-2000 is based on the notion of the viewpoint of a stakeholder with a set of concerns, and it defines view, architectural description, architecture and system accordingly. However, despite the fact that this approach has led to a useful reconsideration of the concepts used in architecture, the drawback is that it does not lead to concepts which are precisely defined in a mathematical sense, and consequently it is neither very clear how to bridge the gap between architectural descriptions and business process models, nor how to incorporate the architectural concepts in tools.

In this paper we study the following two research questions.

1. How to incorporate business process models in enterprise architectures to analyze and simulate their behavior?

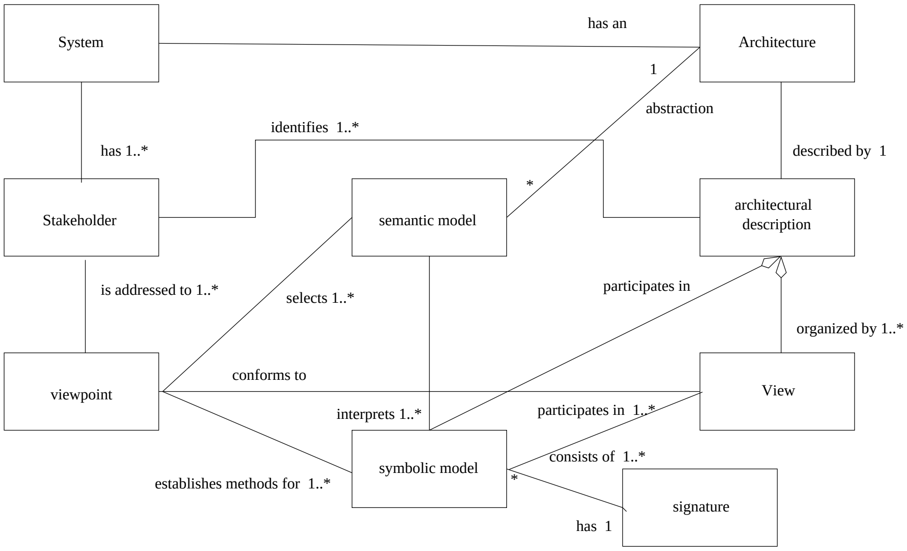


Figure 9.1: Extension of IEEE with LVA

2. How to provide architectural tool support for enterprise architectures with business process models?

To incorporate business process models in enterprise architectures, we believe that we have to extend the IEEE architectural concepts. Consider the main IEEE 1471 standard definitions.

architecture: The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.

system: A collection of components organized to accomplish a specific function or set of functions.

system stakeholder: An individual, team, or organization (or classes thereof) with interests in, or concerns relative to, a system.

view: A representation of a whole system from the perspective of a related set of concerns.

viewpoint: A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis.

These definitions do not reflect the distinction between enterprise architectures and business process models. Our extension of the IEEE conceptual model is visualized in Figure 9.1, in which a symbolic model corresponds to the IEEE concept of model, and which contains the two new concepts semantic model and signature (we leave out IEEE 1471-2000 concepts not related to our new concepts).

Semantic model. The missing concept in the IEEE 1471-2000 to bridge the gap between enterprise architectures and business process models is the notion of a semantic model, which *interprets* symbolic models.

Signature of an architecture. Moreover, each symbolic model has a signature, which contains besides the usual concepts and relations (including special relations like is-a) also functions. The functions play a crucial role in our proces models, as some of them are interpreted by *actions* in the semantic model.

Finally, in contrast to IEEE 1471-2000 we distinguish between the conceptualization of an architecture and its vizualization (though this is not visualized in Figure 9.1).

Concerning tool support, our logical viewpoint provides the formal foundations for the use of XML as a representation language for the signature of an architecture, and more generally as a representation langauge for symbolic as well as semantic models. In this paper we use AML instead of XML, which is equivalent with XML, but designed to be better readable for humans. Roughly, in AML the end tags and angle brackets are replaced by indentation principles.

Moreover, we promote the use of the Rule Markup Language or RML as a language to describe model transformations and thus actions. As explained in detail in this paper, actions are interpreted as functions, and can thus be described by their input/output behavior, which can be described by transoformation rules. RML consists of a small set of XML constructs that can be added to an existing XML vocabulary in order to define RML rules for that XML vocabulary. These rules can then be executed by RML tools to transform the input XML according to the rule definition.

The layout of this paper is as follows. In Section 9.2 we introduce a running example to explain our definitions. In Section 9.2.1 we explain the signature, the distinction between symbolic and semantic model, and the actions. In Section 9.5 we discuss tool support, XML, AML and RML.

9.2 Archimate: a running example

Archimate is an enterprise architecture modelling language [JvBA$^+$03, ea04]. It provides through a metamodel concepts for architectural design at a very general level, covering for example the business, the application, and the technology architecture of a system. The Archmate language resemble the business language Testbed [EJL$^+$99] but it has also a UML-flavour, introducing concepts like interfaces, services, roles and collaborations.

In the remainder of this paper, we will consider as running example, the enterprise architecture of a small company, called *ArchiSell*, modeled using the Archimate language. In *ArchiSell*, employees sell products to customers. The products are delivered to ArchiSell by various suppliers. Employees of ArchiSell are responsible for ordering products and for selling them. Once products are delivered to ArchiSell, each product is assigned an owner, responsible for selling the product.

To describe this enterpise we will need the ArchiMate meta-concepts and their relationships as presented in Figure 9.2. In particular, we will use structural concepts (product, role and object) and structural relationships (association), but also a behavioural concepts (process) and behavioural relationships (triggering). Behavioural and structural concepts are connected by means of the assignment and access relationships.

A product is a phisical entity that can be associated with roles. A role is the representation of a collection of responsability that may be fulfilled some entity capable of performing behaviour. The assignment relation links processes with the roles that perform them. The triggering relation between process describes the temporal relations between them. When executed, process may need to access data, whose representation is here called object.

We will specifically look at the *business process architecture* for ordering products, depicted in Figure 9.3.

In order to fulfill the business process for ordering a product, the employee has to perform the following activities:

- Before placing an order, an employee must register the order within the Order Registry.

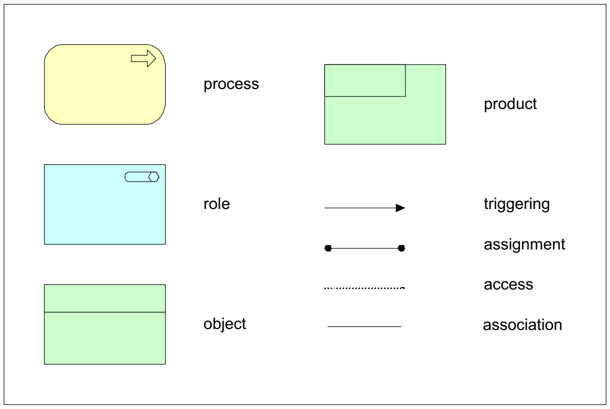


Figure 9.2: Some concepts and relations

- After that, the employee places the order with the supplier.
- As soon as the supplier delivers the product(s), the employee first checks if there is an order that refers to this delivery. Then, he/she accepts the product(s).
- Next, the employee registers the acceptance of the product(s) within the Product Registry and determines which employee will be the owner of the product(s).

9.2.1 Systems and architectures

Following IEEE 1471-2000, every system has an architecture. In our logical perspective which abstracts from pragmatics, like design principles, an architecture is the structure and dynamics of a system consisting of its components and their relationships.

The architecture of a system is purely conceptual and different from particular symbolic descriptions of that architecture. An architectural description consists of several symbolic models (also called model in [Soc00]) and other pragmatic information. Examples of the latter are the architectural rationale. In the next sections we focus on the logical nature of these symbolic models which involves their syntax and semantics.

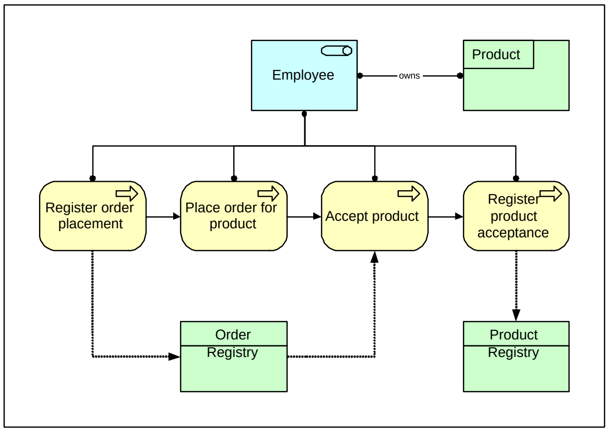


Figure 9.3: A Business Process Architecture

The signature of an architecture

The very core of a symbolic model of an architecture consists of its *signature* which specifies its *name space*. The names of a signature are used to denote symbolically the structural elements of the architecture, their relationships, and their dynamics. The *nature* of each structural element is specified by a *sort*, and each architectural relationship by a *relation* between sorts. Additionally, a signature includes an ordering on its sorts and its relations for the specification of a classification in terms of a generalization relation on the structural elements and the architectural relations. For example, the sort object in Figure 9.2 can be defined as a generalization of both the sorts Order_Registry and Product_Registry given in Figure 9.3, to indicate that every element in Order_Registry or Product_Registry is also an element of sort object. Also, an association between role and product is a generalization of the relation owns between Employee and Product.

The ordering on sorts and relations is in general used to capture certain aspects of the *ontology* of an architecture. Other ontological aspects can be captured by the aggregation and containment relations. For technical convenience however we restrict to the generalization relation only.

Definition 8 *A signature consists of*

- *a partially ordered set of* primitive sorts*, also called the* sort hierarchy*;*
- *a partially ordered set of* relations*, where each relation is of the form*

$R(S_1, \ldots, S_n)$, with R the name of the n-ary relation and S_i the primitive sort of its ith argument.

We allow *overloading* of relation names, i.e., the same name can be used for different relations. For instance, given the primitive sorts *Person*, *Boss*, and *Employee*, the relations $Responsible(Boss, Employee)$ and $Responsible(Person, Person)$ are in general two different relations with the same name.

Further information about the architecture is expressed symbolically in terms of suitable extensions of one of its signatures. Usually a signature is extended with operations for constructing complex *types* from the primitive sorts. Examples are the standard type operations like *product type*

$$T_1 \times T_2$$

of the types T_1 and T_2, and the *function type*

$$T_1 \to T_2$$

of all functions which require an argument of type T_1 and provide a result of type T_2. Note that a relation $R(S_1, \ldots, S_n)$ is a sub-type of $S_1 \times \cdots \times S_n$.

Given functional types, the name space of a signature can be extended with *functions*

$$F(T_1){:}T_2,$$

where F specifies the name of a function of type $T_1 \to T_2$. Functions can be used to specify the *attributes* of a sort. For example, given the primitive sorts *Employee* and $\mathbb{N}$, the function $Age(Employee){:}\mathbb{N}$ is intended for specifying the age of each person.

Note that *multi-valued* functions

$$F(T_1, \ldots, T_n){:}T'_1, \ldots, T'_m$$

can be specified by the functional type $T \to T'$, where T denotes the product type $T_1 \times \cdots \times T_n$ and T' denotes the product type $T_1 \times \cdots \times T_n$. In general, functions are also used to specify symbolically the dynamics of an architecture.

The next example shows the signature of the business process architecture described in Figure 9.3. It is written in AML, a human-understandable notation for generating XML documents. AML and the corresponding tool-support will be discussed in Section 9.5.

Example 2 *The sorts of the example described in Figure 9.3 are simply enumerated in AML by*

```
Role
Object
Employee
Product
product
Order_Registry
Product_Registry
```

Note that we did not include processes as a sort (in our logical view explained above, processes are modeled as functions). The subsort relation is specified in AML by the following enumeration

```
is-a
    domain name=Employee
    codomain name=Role
is-a
    domain name=Order_Registry
    codomain name=Object
is-a
    domain name=Product_Registry
    codomain name=Object
is-a
    domain name=owns
    codomain name=association
```

Note that we have encoded meta-model information of an architecture as part of the signature of the architecture itself. The relation between the meta-model sorts and relations and architectural sorts and relations is expressed by the respective partial orders between sorts and relations of the signature.

In AML the owns-relation itself is specified by

```
 owns
    domain name=Employee
    codomain name=Product
```

Finally, the processes are specified in AML as functions. The types of the arguments and result values are determined as follows: A role which is assigned to a process specifies the type of both an argument and a result value of the corresponding function. Similarly, an outgoing access relation from a process to an object specifies the type of both an argument and a result value of the corresponding function. On the other hand, an incoming access relation from an object to a process only specifies the type of the corresponding argument (this captures the property of 'read-only').

```
Register_order_placement
      domain name=Employee
      domain name=Order_Registry
      codomain name=Employee
      codomain name=Order_Registry
Place_order_for_product
      domain name=Employee
      codomain name=Employee
Accept_product
      domain name=Employee
      domain name=Order_Registry
      codomain name=Employee
Register_product_acceptance
      domain name=Employee
      domain name=Product_Registry
      codomain name=Employee
      codomain name=Product_Registry
```

Note that the triggering relation is not included in our concept of a siganture. In our view such a relation specifies a temporal ordering between the processes which is part of the business process language discussed below in section 9.3.

The recommendation IEEE 1471-2000 [Soc00] emphasizes that views on an architecture should be seen from the perspective of a viewpoint of a stakeholder, that has several concerns. In our logical characterization, a viewpoint is essentially a partial transformation over signatures, and a view is a visualization of the result of the transformation, given a visualization.

Summarizing, the signature of an architecture focuses on the symbolic representation of the structural elements of an architecture and their relationships, abstracting from other architectural aspects like rationale, pragmatics and visualization. It emphasizes a separation of concerns which allows to master the complexity of the architecture. Notably, the signature of an architecture can be easily formalized in XML for storage and communication purpose, and can be integrated as an independent module with other tools including, e.g., graphics for visualization. In the following sections we define the formal *semantics* of a symbolic model of an architecture. Such a semantics provides a formal basis for the development and application of tools for the *logical analysis* of the dynamics of an architecture.

Interpretation of Types

In this section we first define a formal interpretation of the types underlying a symbolic model.

Definition 9 *An* interpretation I *of the types of a signature assigns to each primitive sort* S *a set* $I(S)$ *of* individuals *of sort* S *which respects the subsort ordering: if* S_1 *is a subsort of* S_2 *then* $I(S_1)$ *is a* subset *of* $I(S_2)$.

Any primitive sort is interpreted by a subset of a universe which is given by the union of the interpretation of all primitive sorts. The hierarchy between primitive sorts is expressed by the subset relation.

An interpretation I of the primitive sorts of a signature of an architecture can be inductively extended to an interpretation of more complex types. For example, an interpretation of the product type

$$T_1 \times T_2$$

is given by the cartesion product

$$I(T_1) \times I(T_2)$$

of the sets $I(T_1)$ and $I(T_2)$. The interpretation of the function type $T_1 \rightarrow T_2$ as the set

$$I(T_1) \rightarrow I(T_2)$$

of all functions from $I(T)1)$ to $I(T_2)$, however, does not take into account the *contra-variant* nature of the function space. For example, since the sort $\mathbb{N}$ of natural numbers is a sub-sort of the real numbers $\mathbb{R}$, a function from $\mathbb{R}$ to $\mathbb{R}$ dividing a real number by 2 is also a function from $\mathbb{N}$ to $\mathbb{R}$, but, clearly, the set of all functions from $I(\mathbb{R})$ to $I(\mathbb{R})$ is *not* a subset of the set of functions from $I(\mathbb{N})$ to $I(\mathbb{R})$. Therefore, given the universe $\mathbb{U}$ defined as the union of all the interpretations of the primitive sorts, we define the interpretation of the function type $T_1 \rightarrow T_2$ by

$$I(T_1 \rightarrow T_2) = \{f \in \mathbb{U} \rightarrow \mathbb{U} \mid f(I(T_1)) \subseteq I(T_2)\}.$$

The function type $T_1 \rightarrow T_2$ thus denotes the set of all functions from the universe to itself such that the image of $I(T_1)$ is contained in $I(T_2)$. Note that if T_1' is a subtype of T_1 and T_2 is a subtype of T_2' then $I(T_1 \rightarrow T_2)$ is indeed a subset of $I(T_1' \rightarrow T_2')$.

In general, there can be a large number of different interpretations for a signature. This reflects the intuition that there are many possible architectures that fit a specific architectural description. In fact, a signature of an architecture basically only specifies the basic concepts by means of which the architecture is described.

9.3 Semantic models

In our logical perspective, a semantic model is a formal *abstraction* of the architecture of a system. The logical perspective presented until now, only concerned the symbolic representation of an architecture by means of its signature. Next we show how to obtain a formal model of a system as a semantic interpretation of the symbolic model of its architectural description.

The semantic model of a system involves its concrete components and their concrete relationships which may change in time because of the dynamic behavior of a system. To refer to the concrete situation of a system we have to extend its signature with names for referring to the individuals of the types and relations. For a symbolic model, we denote by $n{:}T$ a name n which ranges over individuals of type T.

Given a symbolic model of an architecture extended with individual names and an interpretation I of its types, we define a semantic model Σ as a function which provides the following interpretation of the name space of the symbolic model covering its relations, functions, and individuals.

Relations For each relation $R(S_1, \ldots, S_n)$ we have a relation

$$\Sigma(R) \subseteq I(S_1 \times \cdots \times S_n)$$

respecting the ordering between relations, meaning that if R_1 is a subrelation of R_2 then $\Sigma(R_1)$ is a subset of $\Sigma(R_2)$.

Functions For each symbolic function $F(T_1){:}T_2$ we have a function

$$\Sigma(F) \in I(T_1 \rightarrow T_2).$$

Variables For each individual name $n{:}S$ we have an element

$$\Sigma(n) \in I(S).$$

Example 3 *For our running example we introduce the following semantic model. In this model we have only two products p1 and p2. This is specified in AML by*

```
Product
    p1
    p2
```

In order to model the processing of orders and products individuals of the sort Employee have a product attribute and an order attribute. These attributes indicate the order and product the employee is managing. These attributes can also be viewed as providing an interface *to the environment consisting of the clients and suppliers. Both the order of a client and the product of a supplier will be stored by an employee (not necessarily the same employee). In our model individuals of the sort Employee are fully charecterized by these attributes. Therefore in our model the sort Employee contains four elements, as described in AML by*

```
Employee
      e1 order=p1 product=p1
      e2 order=p1 product=p2
      e3 order=p2 product=p1
      e4 order=p2 product=p2
```

In our simple model both the Order_Registry and Product_Registry can contain only information about one of the two products p1 and p2 (in section 9.5 we discuss how to model an Order_Registry as a finite list of orders). Consequently, we can identify in this simple model the interpreation of these sorts with that of Product:

```
Order_Registry
     p1
     p2

Product_Registry
     p1
     p2
```

The interpretation of the processes of our running example in this model are specified in AML by means of matrices of input/output pairs. For example, in the following we illustrate two such input/output pairs belonging to the interpretation of Register_order_placement: it replaces the product stored in the Order_Registry by the product stored in the order of the employee:

```
matrix function=Register_order_placement
    input
```

```
        e1 order=p1 product=p1
        p1
    output
        e1 order=p1 product=p1
        p1
    input
        e1 order=p1 product=p2
        p2
    output
        e1 order=p1 product=p2
        p2
```

The other processes are formally described in a similar manner. Because of space limititaion we restrict to a informal description of their interpreations.

The function Place_order_for_product does not affect the information stored in an employee (in more refined models this function may in fact describe an update which records informationm about the supplier involved).

The function Accept_product simply checks whether the product managed by an employee is stored in the Order_registry. We model this check as a partial *function which contains only those input/output pairs for which the product stored in the Order_registry coincides with the product managed by the employee. Note that the product managed by the employee results from the delivery of a supplier and that the order managed by an employee may have changed after it has been stored in the Order_registry.*

The function Register_product_acceptance simply stores the product managed by the employee in the Product_registry.

Finally, in order to refer to the elements of the different sorts we introduce individual names emp:Employee, order-reg: Order_Registry, and product-reg: Product_Registry. A semantic model assigns individuals to these names, for example, such an assignment is specified in AML simply by

```
   emp = e1 order=p1 product=p1
   order-reg= p1
   product-reg= p2
```

Note that this assignment describes an employee which manages an order of product p1 and a delivery of product p1, an Order_registry which registers an order of product p1, and a Product_registry which registers the acceptance of a product p2.

Dynamics of a system

The dynamics of a concrete system with an architectural description given by its signature can be specified in different ways. Below we distinguish two different use of functions to describe the dynamics of a system: one where functions are seen as primitive actions that change the state of a system, and another where functions are seen as data transformers.

In the first case, we define the *action of a function* $F(S){:}T$ by an assignment of the form

$$n{:} = F(m)$$

where $n{:}T$ and $m{:}S$ are names ranging over the types T and S, respectively. The execution of such an action in a semantic model Σ *assigns* to the name n the return value of

$$\Sigma(F)(\Sigma(m))$$

which denotes the result of applying the function $\Sigma(F) \in I(S \to T)$ to the element $\Sigma(m) \in I(S)$. Note that actions transform semantic models (i.e. the state of a system) but not the interpretation of a signature (i.e. the structural information of a system).

Example 4 *Given the interpretation of the individual names e and or of the example 3, the execution of the action*

$$\text{e,or:=Register\_order\_placement(e,or)}$$

results in the new semantic model Σ' *such that* $\Sigma'(\text{or}) = \text{p1}$.

Given this concept of an action as a transformation of semantic models, we can define more complex *processes* by combining actions, that is, we can define operations on actions determining the order of their execution. For example, we can define the sequential composition $n{:} = F(m);n'{:} = G(m')$ of two actions $n{:} = F(m)$ and $n'{:} = G(m')$ as the composition their transformation of semantic models. Other operations on actions include case structure, loops, parallel composition, and synchronization.

Example 5 *Given the above sorts Product and Employee, and a function name Produce of type Employee × Product → Product, we can define a* pipeline *by*

$$\text{p1:=Produce(e1,p1);p2:=Produce(e2,p1)},$$

where e1 and e2 denote individual employees and p1 and p2 denote some products.

The above interpretation of functions as actions forms a formal basis for the introduction of process algebras and corresponding analysis techniques in business process modeling. A process algebra [Hoa85] is a structured approach for constructing complex processes out of actions. Alternatively, we can use functions to specify the *data-flow* in a system illustrating how data is processed in terms of inputs and outputs. In this view a multi-valued function

$$F(T_1, \ldots, T_n){:}T'_1, \ldots, T'_m$$

is interpreted as an *asynchronous* process transforming data as follows. It has an *input channel* for each of its arguments; when on each input channel data, i.e., an element of the corresponding type, has arrived it *outputs* the result values on corresponding *output channels*. Such processes can be connected via their channels in a *data-flow network* [Kah74] pictorially represented by a Data Flow Diagram [GS79]. Because of space limitiations we omit the formal details.

9.4 Design support

In this section we discuss the support that can be offered by our logicla perspective to describe the evolution of a system. In particular we will briefly describe the role of logical languages and design action in the design of an architecture.

Logical languages

Logical extension of a signature consists in considering types as predicate symbols that can be combined into more complex formulae by means of logical operators like conjunction and disjunction. The resulting logical language can be used to constraint the set of semantic models under consideration. There are several logical languages that can be used as logical extensions of a signature, and a more detailed description of them is beyond the scope of this paper. We just mention here description logics [BCM03] as formalism for constrining semantic models and for reasoning about architecture. They are tailored towards a representation of architecture in terms of concepts and relationships between them. A description logic system consists of the following components:

1. a description language to construct complex description from simple ones;
2. a specification formalism to make statements about how concepts and relations are related each other (TBox) or to make assertions about individuals (ABox)
3. a reasoning procedure.

The advantage of usimg description logics is that they can be formulated in terms of digrams, called the Entity-Relationships Diagrams (ERD) [Che76]. Basically they illustrate the logical structure of a system in terms of concepts and their relationships.

Temporal logics [MP92] are specially tailored towards the specification of the dynamic aspects of a systems. They consists of some atomic predicates on the semantic models together with the prositional connectives and some temporal opeartors like next (X), until (U), some time in the future (F), and always in the future (G). In our view, a temporal logic is intepreted ‘ in terms of sequences of semantics models generated by the actions of the symbolic model. For example the formula

```
emp.order = p1
implies
(emp.order = p1 U order_reg = p1)
```

specifies that if employee emp has received an order for product p1, then eventually the order will be register and until then the employee cannot process any different order.

Design actions

A design action is a transformation between symbolic model. It contains some additional non-logical information that can used to describe the evolution of the system. Examples are actions for adding sorts or relations, for deleting them, or for renaming them. Design actions can be realised by means of rules (for example expressed in RML) that have as antecedent a set of parameter and as consequence a description of the change. When the parameters are collected the rule can fire resulting in a new symbolic model as described in consequence of the rule.

9.5 Tool support

In this section we discuss how our logical perspective provides a formal basis for the integration of XML based tools for the semantic analysis of architectures.

The Extensible Markup Language (XML) [XML] is a universal format for documents containing structured information so that they can be used over the internet for web site content and several kinds of web services. It allows developers to easily describe and deliver rich, structured data from any application in a standard, consistent way. Today, XML can be considered as the lingua franca in computer industry, increasing interoperability and extensibility of several applications. Terseness and human-understandability of XML documents is of minimal importance, since XML documents are mostly created by applications for importing or exporting data.

The ASCII Markup Language (AML) [Jaca] used in this paper is an alternative for XML syntax. AML is designed to be concise and elegant and easy to use. AML uses indentation to increase readability and to define the XML tree hierarchy: indentation level corresponds to depth, sometimes called level, in the tree. No indentation is required for the set of attributes that immediately follows each attribute name.

In the next sub-section we describe a tool for transforming XML documents that can be used for analysis of architectural description, and in particular for the definition and simulation of the system behavior.

9.5.1 The Rule Markup Language

RML stands for Rule Markup Language. It consists of a set of XML constructs that can be added to an existing XML vocabulary in order to define RML rules for that XML vocabulary. These rules can then be executed by RML tools to transform the input XML according to the rule definition. The set of RML constructs is concise and shown in Table 2.1.

Rules defined in RML consist of an antecedent and a consequence. The antecedent defines a pattern and variables in the pattern. Without the RML constructs for variables this pattern would consist only of elements from the chosen XML vocabulary. The pattern in the antecedent is matched against the input XML. The variables specified with RML constructs are much like the wildcard patterns like `*` and `+` and `?` as used in well known tools like `grep`, but the RML variables also have a *name* that is used to remember the matching input. Things that can be stored in RML variables are element names, element attributes, whole elements (including the children), and lists of elements.

If the matching of the pattern in the antecedent succeeds then the variables are bound to parts of the input XML and they can be used in the consequence of an RML rule to produce output XML. When one of the RML tools applies a rule to an input then by default the part of the input that matched the antecedent is replaced by the output defined in the consequence of the rule; the input surrounding the matched part is kept intact.

Below we show an example of RML by presenting the rule that defines the state transformation of the action

emp,order-reg:=Register_order_placement(emp,order-reg)

of our running example (emp and order-reg are individual names for an employee and the Order_registry, respectively). Content-preserving RML constructs have been omitted for clarity.

```
div class=rule name="Register order placement"
    div class=antecedent
        variables
            rml-Employee order=rml-OrderName
                product=rml-ProductName
            or
                orders
                    rml-list name=oldOrders
    div class=consequence
        variables
            rml-Employee order=rml-OrderName
                product=rml-ProductName
            or
                orders
                    rml-use name=oldOrders
                    order name=rml-OrderName
```

In the `antecedent` of the rule the matching algorithm first looks for an element with name `variables` which contains that part of the AML representation of the semantic model discussed 3 that stores the values of the names emp (of sort Employee) and order-reg (of sort Register_order) If that is found it looks for children of that element: one child with an `order` and `product` attribute (an employee), and one child with the name `r1` (the order registry). The algorithm binds the employee name emp to RML variable

`Employee` and it binds the values of the `order` and `product` attributes to `OrderName` and `ProductName` respectively. The list of old orders, a list of XML elements that are the children of the `orders` child of the `r1` order registry, is bound to RML variable `oldOrders`. In the consequence of the rule the variables are reused in the output and an order element with the correct name is appended to the `oldOrders` list. Note that by means of this RML rule we have extended the semantic model of our running example to an interpretation of the sort Order_registry of *unbounded* capacity.

We see here that in a straightforward way, thanks to the wildcard matching technique used in RML, a pattern can be matched that is distributed over various parts of the input XML. Such pattern matching is hard to define with other existing approaches to XML transformation because they do not use of the problem domain XML for defining transformation rules: transformations are defined either in special purpose language like the Extensible Stylesheet Language Transformation (XSLT), or they are defined at a lower level by means of programming languages like DOM and SAX.

RML does not define, need, or use another language, it only adds a few constructs to the XML vocabulary used, like the wildcard pattern matching. RML was designed to make the definition of executable XML transformations also possible for other stakeholders than programers. This is of particular relevance when transformations capture for instance business rules. In this way it is possible to extend the original model in the problem domain XML vocabulary with semantics for that language. Similarly, it is also possible to define rules for constraining the models with RML.

9.5.2 RML as a tool for architectural description

As illustrated above, with RML a formal definition can be given of the dynamics of the basic actions of an architecture in terms XML transformations. This allows for a formal use of process algebras [Hoa85] in the modeling and analysis of business processes. In fact, the use of RML allows the formal definition of one own's business process constructs on top of the semantic description of the basic actions.

As a simple example, the execution of an action a by the process P=a.b that specifies a temporal order between a and b (namely, first a and than b), can be described in a process algebra by a *transition* of the form (we abstract from the state)

```
a.b -> b
```

As a transformation in RML this transition can be specified by the following rule:

```
div class=rule
    div class=antecedent
        process name=rml-P
            prefixes
                rml-A
                rml-B
    div class=consequence
        process name=rml-P
            prefixes
                rml-B
```

The removal of the `a` prefix is easily specified in such an RML rule as the removal of an element from a list of children elements.

XML tranformations normally involve creating links between elements by means of cross-referencing attributes, or reordering elements, or adding or removing elements, but does typically not include things like integer arithmetic and floating point calculations. In case of such transformations the RML tool will have to be combined with another tool that can do the desired calculation. For modelling business architectures a transformation that can not be expressed with XML+RML alone is rather uncommon, but they may occur when the user is interested in a simulation of a model. We have applied combinations of RML with other components like programming language interpreters successfully in the EU project OMEGA (IST-2001-33522, URL: http://www-omega.imag.fr) that deals with the formal verification of UML models for software. That tool for the simulation of UML models does the XML transformations with RML, and uses an external interpreter for example for floating point calculations on attributes in the XML encoding.

9.6 Summary and outlook

In this paper we consider the relation between enterprise architectures and much more detailed business process models. The missing link to bridge the gap between the two worlds is the notion of a semantic model in the IEEE 1471-2000 standard [Soc00] . We show how semantic models can be distinguished from the models used within the standard, which we call symbolic model. This distinction provides a formal basis for the introduction of a formal definition and analysis of business processes. Moreover, we extend the IEEE standard with the notion of the signature, which serves as the basis of the enterprise architecture description, as well as the semantic model.

Semantic models are at the center of our logical perspective on enterprise architectures which integrates both static and dynamic aspects. The framework we have developed allows the integration of various models for business processes, ranging from process algebras to data-flow networks.

Furthermore, we have introduced a XML tool for the transformation of XML data and showed how it can be used to simulate business processes.

There is a rich literature of business processes. However, as far as we know, our logical perspective is a first attempt to a formal integration of such processes in enterprise architectures. We believe that our logical framework (plus tool support) also provides an promising basis for the further design and development of business process languages and corresponding tools.

Acknowledgements This paper results from the ArchiMate project (`http://archimate.telin.nl`), a research initiative that aims to provide concepts and techniques to support architects in the visualisation, and analysis of integrated architectures. The ArchiMate consortium consists of ABN AMRO, Stichting Pensioenfonds ABP, the Dutch Tax and Customs Administration, Ordina, Telematica Institute, CWI, University of Nijmegen, and LIACS.

Chapter 10

Using XML Transformation for Enterprise Architecture

Authors: F.S. de Boer, M.M. Bonsangue, J.F. Jacob, A. Stam, L. van der Torre

10.1 Introduction

In this paper, we investigate the use of XML transformation techniques in the context of Enterprise Architectures. We have split up this research question into two subquestions:

- How can we use XML transformation for the generation of views on an architecture (selection and visualization)?
- How can we use XML transformation for the analysis of architectures?

First, we will introduce the reader to the term Enterprise Architecture, the ArchiMate project, and XML. Second, we will give a short overview of our research methodology.

10.1.1 Enterprise Architectures

A definition of Architecture quoted many times is the following IEEE definition: "the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution" [Soc00]. Therefore, we can define Enterprise Architecture [MAS$^+$03] as the Architecture of an enterprise. It covers principles, methods and models for the design and implementation of business processes, information systems, technical infrastructure and organizational structure.

Architectural information is usually contained in Architectural models. With these models and the information they incorporate, stakeholders within an organization are able to get more insight into the working of the organization, the impact of certain changes to it, and ways to improve its functioning.

Usually, we can distinguish between architectural descriptions that cover the as-is situation of an organization and descriptions that cover its intended to-be situation. According to IEEE, views are part of an architectural description. Views conform to viewpoints which are useful for certain stakeholders.

10.1.2 ArchiMate

Within the ArchiMate project, a language for Enterprise Architecture has been developed [JvBA$^+$03]. This language can be used to model all architectural aspects of an organization. An overview of the concepts in the ArchiMate language is given in Figure 10.1.

As can be seen, the language contains concepts for several aspects of an organization. The individual concepts can be specialized for multiple domains, like the business domain, application domain or technical domain. Thus, a *Service* can be a business service, an application service or a technical service, for example.

10.1.3 XML

The Extensible Markup Language (XML) [XML] is a universal format for documents containing structured information so that they can be used over the Internet for web site content and several kinds of web services. It allows developers to easily describe and deliver rich, structured data from any application in a standard, consistent way.

Today, XML can be considered as the lingua franca in computer industry, increasing interoperability and extensibility of several applications. Terseness and human-understandability of XML documents is of minimal importance,

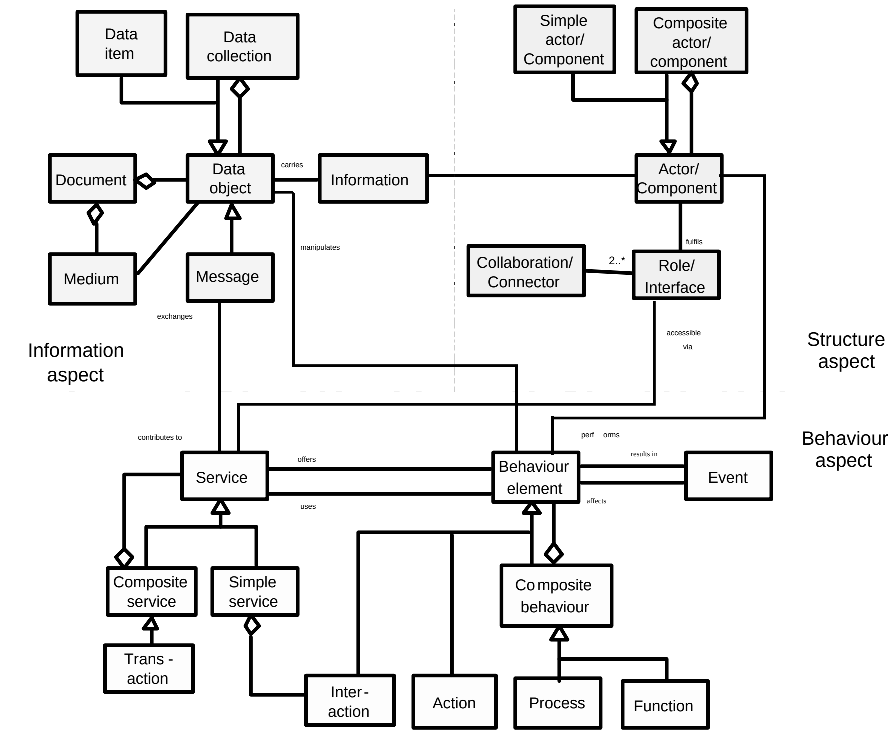


Figure 10.1: The ArchiMate metamodel

since XML documents are mostly created by applications for importing or exporting data.

10.1.4 Research methodology

First, we have developed a running example for verification of our ideas and techniques. During the development, we have refined the research questions as follows:

1. Given a set of architectural information described in a single XML document. How can we use XML transformation to select a subset of this information for a specific architectural view?

2. Is it possible to use XML transformation for *visualization*? I.e. is it possible to transform an XML document containing an architectural

description into another XML document containing visual information in terms of boxes, lines, etc.?

3. How can we use XML transformation to perform analyses on an architectural description? At first, we have chosen to specifically look at a specific form of *impact analysis*: given an entity within the architectural description which is considered to be modified or changed, which other entities in the description are possibly influenced by this change?

The second step consisted of developing an XML document containing the architectural information of the running example. As a basis for the architectural description, we have used an XML Schema containing the concepts from the ArchiMate metamodel.

Thereafter we developed an XML Schema for visualization information and built a *model viewer* which interprets this visualization information and shows this information on the screen. The aim was to keep the model viewer as "dumb" as possible, in order to make full use of XML transformation techniques for the actual visualization.

In the fourth step, we selected a transformation tool, namely the *Rule Markup Language*(RML), and built the transformation rules for selection, visualization and impact analysis.

10.1.5 Document layout

In Section 10.2 the Rule Markup Language (RML) is introduced. In Section 10.3 we will introduce the running example: ArchiSurance, a small insurance company which has the intention to phase out one of its core applications. In Section 10.4 we show transformation rules for selection and visualisation of architectural views, while in Section 10.5 we illustrate transformation techniques for analysis by means of performing a small impact analysis. In Section 10.6 we conclude.

10.2 The Rule Markup Language

RML stands for Rule Markup Language. It consists of a set of XML constructs that can be added to an existing XML vocabulary in order to define RML rules for that XML vocabulary. These rules can then be executed by RML tools to transform the input XML according to the rule definition.

The set of RML constructs is concise and shown in Table 2.1 with a short explanation of the constructs.

Rules defined in RML consist of an antecedent and a consequence. The antecedent defines a pattern and variables in the pattern. Without the RML constructs for variables this pattern would consist only of elements from the chosen XML vocabulary. The pattern in the antecedent is matched against the input XML. The variables specified with RML constructs are much like the wild-card patterns like `*` and `+` and `?` as used in well known tools like `grep`, but the RML variables also have a *name* that is used to remember the matching input. Things that can be stored in RML variables are element names, element attributes, whole elements (including the children), and lists of elements.

If the matching of the pattern in the antecedent succeeds then the variables are bound to parts of the input XML and they can be used in the consequence of an RML rule to produce output XML. When one of the RML tools applies a rule to an input then by default the part of the input that matched the antecedent is replaced by the output defined in the consequence of the rule; the input surrounding the matched part is kept intact.

RML does not define, need, or use another language, it only adds a few constructs to the XML vocabulary used, like the wild-card pattern matching. RML was designed to make the definition of executable XML transformations also possible for other stakeholders than programmers. This is of particular relevance when transformations capture for instance business rules. In this way it is possible to extend the original model in the problem domain XML vocabulary with semantics for that language. Similarly, it is also possible to define rules for constraining the models with RML.

10.2.1 Comparison with other techniques

XSLT is a W3C language for transforming XML documents into other XML documents.

The *RuleML* [com] community is working on a standard for rule-based XML transformations. Their approach differs from the RML approach: RuleML superimposes a special XML vocabulary for rules. This makes the RuleML approach complex and thus difficult to use in certain cases.

The *Relational Meta-Language* [Pet94] is a language that is also called RML, but intended for compiler generation, which is much more roundabout and certainly not usable for rapid application development like with RML in this paper.

Another recent approach is *fxt* [BS02], which, like RML, defines an XML syntax for transformation rules. Important drawbacks of fxt are that it is rather limited in its default possibilities and relies on hooks to the SML programming language for more elaborate transformations.

Other popular academic research topics that could potentially be useful for rule based XML transformations are term rewriting systems and systems based on graph grammars for graph reduction. However, using these tools for XML transformations is a contrived and roundabout way of doing things. To use these kind of systems, there has to be first a translation from the problem XML to the special-purpose data structure of the system. And only then, in the tool–specific format, the semantics is defined. But the techniques used in these systems are interesting, especially for very complex or hard transformations, and it looks worthwhile to see how essential concepts of these techniques can be incorporated in RML in the future.

Compared with the related work mentioned above, a distinguishing feature of the RML approach is that RML *re-uses* the language of the problem itself for matching patterns and generating output. This leads in a natural way to a much more usable and clearly defined set of rule based transformation definitions, and an accompanying set of tools that is being used successfully in practice.

10.3 Running Example

Throughout this paper, we will use a running example to illustrate our ideas. The architectural description of this example can be found in the models in Figures 10.2, 10.3, 10.4 and 10.5.

A small company, named ArchiSurance, sells insurance products to customers. Figure 10.2 contains a Business View of the company. Two *roles* are involved, namely the insurance company and the customer, which work together in two *collaborations*, namely negotiation, i.e. the set of activities performed in order to come to an appropriate insurance for a customer by discussion and consultation, and contracting, i.e. the set of activities performed in order to register a new customer and let it sign a contract for an insurance policy.

Within Figure 10.3, the business *process* for selling an insurance product to a customer is shown, together with the *roles* and/or *collaborations* that

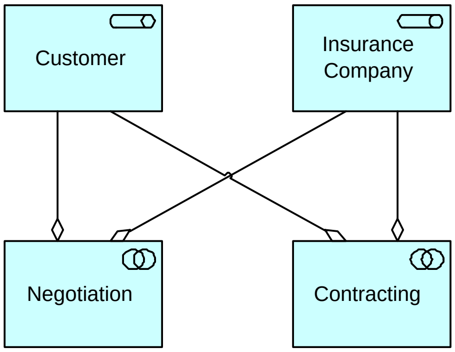


Figure 10.2: a Business View of ArchiSurance

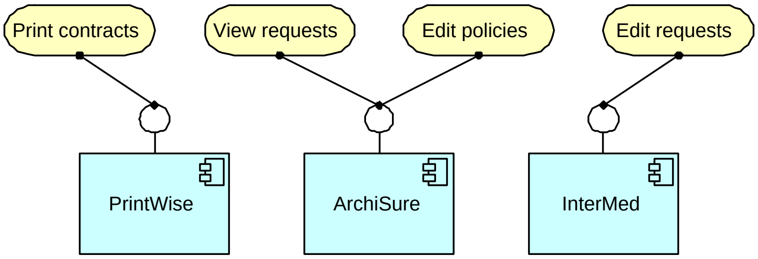


Figure 10.3: a Process View of ArchiSurance

are involved in executing the individual steps within the process.

Figure 10.4 shows the software products (*components*) that are used within the ArchiSurance company and the *services* they offer. *ArchiSure* is a custom-made software application for the administration of insurance products, customers and premium collecting. *PrintWise* is a out-of-the-box tool for official document layout and printing. *Intermed* is an old application, originally meant for intermediaries to have the possibility to enter formal requests for insurance products for their customers. The application is now used by employees of the insurance company, since no intermediaries are involved in selling insurance products anymore. Actually, the company would like to phase out this application.

In Figure 10.5, the *process* for selling products is shown again, together with the *services* that are used within each step.

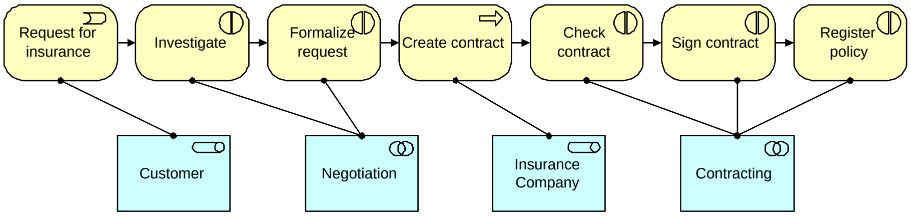


Figure 10.4: an Application View of ArchiSurance

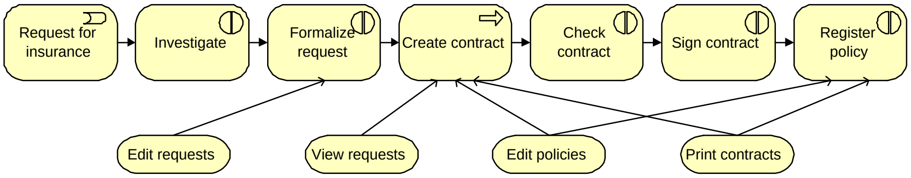


Figure 10.5: a Service View of ArchiSurance

10.3.1 An XML description of the example

Though the four views are depicted separately, they are clearly related to each other via the concepts they contain. In this small example, it is possible to imagine the *big picture* in which all ArchiSurance information is contained.

Within the ArchiMate project, a XML Schema has been developed which can be used for storage or exchange of architectural descriptions. Based on this Schema, we have created an XML document that contains all information about ArchiSurance. For illustration, a fragment of this XML document is shown below. It contains the XML equivalent of Figure 10.2.

```
<role id="002" name="Customer"/>
<role id="003" name="Insurance Company"/>
<collaboration id="004" name="Negotiation"/>
<collaboration id="006" name="Contracting"/>
<composition id="035" name="composition">
    <from href="004"/>
    <to href="002"/>
</composition>
<composition id="036" name="composition">
    <from href="004"/>
    <to href="003"/>
</composition>
<composition id="041" name="composition">
    <from href="006"/>
    <to href="002"/>
</composition>
<composition id="042" name="composition">
```

```
    <from href="006"/>
    <to href="003"/>
</composition>
```

10.4 Selection and Visualisation

The initial XML document contains the concepts and relations based on the ArchiMate metamodel. It does not contain information about which concepts are relevant for which views, nor does it describe how to visualize the concepts. We can use RML rules for both tasks, as will be illustrated in the following sections.

10.4.1 Selection

Within a single view, usually a selection of the entire set of concepts is made. For example, the Business View in our example only contains *roles* and *collaborations*. For this purpose, RML rules have to filter out all unnecessary information from the XML document and thus create a new document that only contains those concepts and relations that are relevant for the view.

We have created the following recipe for selection:

1. add a specific *selection* element to the XML document which is going to contain the selected concepts;

2. iterate over the document and move all relevant concepts into the specific *selection* element;

3. iterate over the document and move all relevant relations into the specific *selection* element;

4. remove all relations within the *selection* element that have one "dangling" end, i.e. that are related at one side to a concept that does not belong to the selection;

5. remove all elements outside the *selection* element.

Note that the step for removing relations with one "dangling" end out of the selection is necessary, because one relation type (e.g. association) can be defined between several different concept types.

The following RML rule illustrates the way all instances of a specific concept are included in the selection:

```
<div class="rule">
    <div class="antecedent">
        <model>
            <rml-list name="list1"/>
            <collaboration rml-others="other">
                <rml-list name="childs"/>
            </collaboration>
            <rml-list name="list2"/>
            <selection>
                <rml-list name="selection"/>
            </selection>
            <rml-list name="list3"/>
        </model>
    </div>
    <div class="consequence">
        <model>
            <rml-use name="list1"/>
            <rml-use name="list2"/>
            <rml-use name="list3"/>
            <selection>
                <rml-use name="selection"/>
                <collaboration rml-others="other">
                    <rml-use name="childs"/>
                </collaboration>
            </selection>
        </model>
    </div>
</div>
```

10.4.2 Visualization

As is described in the introduction, we wanted to keep the model viewer as “dumb” as possible, in order to illustrate the way in which XML transformations can be used for creating several visualizations for a single XML document.

For this purpose, we have made a specific XML schema which can be interpreted by the model viewer without having to know anything about the ArchiMate language. The following XML fragment illustrates this language.

```
<container height="80" id="014" type="interaction" width="100" >
      <box color="khaki1" height="80" type="round" width="100" x="0" y="0" z="0" />
      <label fieldname="name" halign="center" text="register policy" x="50" y="40" z="1" />
      <icon height="15" type="splitcircle" width="15" x="75" y="10" z="1" />
</container>
```

```
<container height="80" id="013" type="interaction" width="100" >
      <box color="khaki1" height="80" type="round" width="100" x="0" y="0" z="0" />
      <label fieldname="name" halign="center" text="sign contract" x="50" y="40" z="1" />
      <icon height="15" type="splitcircle" width="15" x="75" y="10" z="1" />
</container>

<arrow from="013" id="020" to="014" type="triggering" >
      <line type="solid" width="1" z="0" />
      <headarrowtip size="10" type="filledarrow" z="1" />
</arrow>
```

The intermediate visualization language has two main constructs: containers and arrows.

Containers are rectangular areas in which several visual elements can be placed. The exact location of those visual elements can be defined relative to the size and position of the container. Each container has a unique identifier which can be used to refer to the original elements in the architectural description.

Arrows are linear directed elements. They have a head and a tail, which both have to be connected to containers (via their identifiers). They also have unique identifiers themselves.

In the example above, two containers and one arrow are defined. In Figure 10.6 the output of the interpretation of this XML fragment by the model viewer is shown. As can be seen in the XML fragment, some visual elements, like "split circle", are built into the model viewer. This has mainly been done for reasons of efficiency.

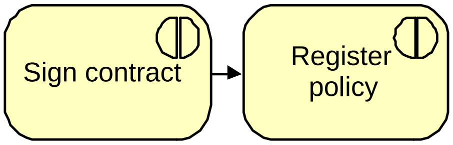


Figure 10.6: Example of the visualization technique used

For the transformation of the original XML model to the visualization information, we have created scripts that transform each concept into its corresponding visualization. An example is given below.

```
<div class="rule">
    <div class="antecedent">
        <interaction id="rml-id" name="rml-name" color="rml-color"/>
    </div>
    <div class="consequence">
        <container id="rml-id" type="interaction" width="100" height="80" color="rml-color">
            <box x="0" y="0" z="0" width="100" height="80" color="khaki1" type="round"/>
            <label x="50" y="40" z="1" halign="center" text="rml-name" fieldname="name"/>
            <icon x="75" y="10" z="1" width="15" height="15" type="splitcircle"/>
        </container>
    </div>
</div>
```

This example rule transforms an *interaction* concept into a visual representation.

The technique presented here is quite powerful: from the same architectural description, it is possible to define different visualization styles, like ArchiMate, UML, etc. In the context of enterprise architectures, this is especially useful since architects often want to have their own style of visualization (for cultural and communication reasons within organizations), without having to conform to a standard defined outside the organization.

10.5 Analysis

Next to selection and visualisation, we have investigated ways to use XML transformation for analysis of enterprise architectures. Our aim was to create a technique for impact analysis, i.e. given an entity within the architectural description which is considered to change, which other entities are possibly influenced by this change?

We have created the following recipe for this analysis:

1. add a specific *selection* element to the XML document which is going to contain the concepts that are considered to be possibly influenced;

2. add a special attribute to the element describing the entity under consideration, which can be used for for example visualisation (in order to make it have a red color, for example);

3. make the element describing the entity under consideration a child of the *selection* element;

4. iterate over all relations included in the analysis and, if appropriate, add a special attribute to them and make them a child of the *selection* element;

5. iterate over all concepts and, if appropriate, add a special attribute to them and make them a child of the *selection* element;

6. repeat the previous two steps until the output is stable;

7. remove the *selection* element, so that we have one list of concepts and relations, of which some concepts have a special attribute which indicates that the change possibly has impact on them.

An example of the output of the analysis is given below. The component "InterMed" is considered to change. It has two new attributes. The *selected* attribute indicates that it belongs to the entities which are possibly influenced by the change, while the *special* attribute indicates that this entity is the unique entity considered to change. The remaining elements describe concepts and relations that are all selected, because they are directly or indirectly related to the "InterMed" component.

```
<component id="082" name="InterMed" selected="yes" special="yes"/>

<composition id="104" name="composition" selected="yes" >
    <from href="082" />
    <to href="094" />
</composition>

<interface id="094" name="Interface" selected="yes" />

<assignment id="112" name="assignment" selected="yes" >
    <from href="094" />
    <to href="090" />
</assignment>

<service id="090" name="edit requests"  selected="yes" />
```

Within Figure 10.7 and Figure 10.8, the output of the model viewer is given for two views. The change of color is done by the visualisation scripts, based on the attributes added during the analysis.

10.6 Summary

Our conclusions about each research question are as follows:

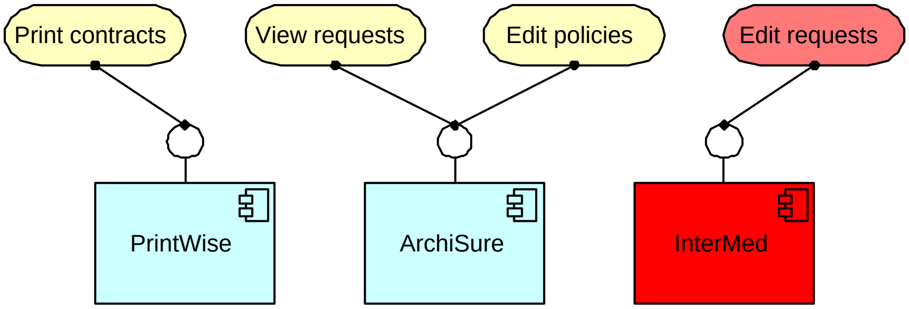


Figure 10.7: The Application View with a selected InterMed application

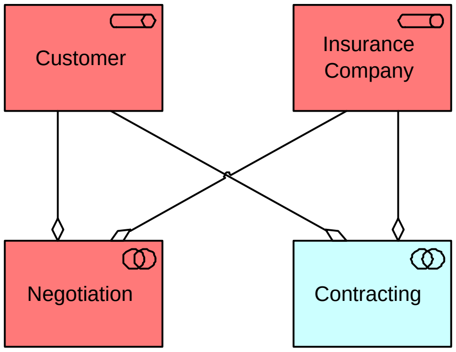


Figure 10.8: The resulting Business View after the impact analysis

10.6.1 Question 1

How can we use XML transformation to select a subset of a set of architectural information for a specific architectural view? In Section 10.4 we have illustrated a way to filter out certain concepts and create a new XML document containing a selection out of the original document.

10.6.2 Question 2

Is it possible to use XML transformation for *visualization*? In Section 10.4 we have shortly described an “intermediate” language for visualization information and illustrated how we can transform an ArchiMate XML document into a Visualization XML document.

10.6.3 Question 3

How can we use XML transformation to perform a specific form of impact analysis on an architectural description? A technique to perform this specific analysis is described in Section 10.5.

10.6.4 Conclusions

The research reported on in this paper shows promising results. The use of XML and transformation techniques for it has several benefits: XML is well-known, the transformation techniques are generic and tools for it are improving rapidly. Transformation rules are well understandable and can be adapted quickly for specific needs or purposes.

The use of XML transformations for visualization proves to be specifically interesting: in many cases, enterprise architecture tools have a fixed way of visualizing information, which hinders architects in representing information in the way they want to. By separating the "visualization step" from the "viewer", architects gain much often demanded flexibility.

Acknowledgments This paper results from the ArchiMate[1] project, a research initiative that aims to provide concepts and techniques to support architects in the visualization and analysis of integrated architectures. The ArchiMate consortium consists of ABN AMRO, Stichting Pensioenfonds ABP, the Dutch Tax and Customs Administration, Ordina, Telematica Institute, CWI, University of Nijmegen, and LIACS.

[1] (`http://archimate.telin.nl`)

Bibliography

[7] Health Level 7. Organization web site at http://www.hl7.org.

[ABdB00] Farhad Arbab, Marcello M. Bonsangue, and Frank S. de Boer. A coordination language for mobile components. In *SAC '00: Proceedings of the 2000 ACM symposium on Applied computing*, pages 166–173, New York, NY, USA, 2000. ACM Press.

[AdBB00] Farhad Arbab, Frank S. de Boer, and Marcello M. Bonsangue. A Logical Interface Description Language for Components. In *COORDINATION '00: Proceedings of the 4th International Conference on Coordination Languages and Models*, pages 249–266, London, UK, 2000. Springer-Verlag.

[AG97] R. Allen and D. Garlan. A Formal Basis for Architectural Connections. *ACM Transactions of Software Engineering and Methodology*, 6(3):213–249, 1997.

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[Arb96] Farhad Arbab. Manifold Version 2: Language Reference Manual. Technical Report ISSN 0169-118X, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1996.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14(3):329–366, 2004.

[Arc] The Archimate project. Website at http://www.telin.nl/-NetworkedBusiness/Archimate/ENindex.htm.

[AwJS96] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs, 2nd edition.* The MIT Press, 1996.

[BCM03] *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2003.

[BDJ+03] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery, 2003.

[BGM01] M. Bozga, S. Graf, and L. Mounier. Automated Validation of Distributed Software using the IF Environment. In Workshop on Software Model-Checking, volume 55. TCS, 2001.

[BH89] Ferenc Belina and Dieter Hogrefe. The CCITT-Specification and Description Language SDL. *Computer Networks*, 16:311–341, 1989.

[BS01] M. Broy and K. Stolen. Specification and Development of Interactive Systems: FOCUS on Streams. In *Interfaces and Refinement.* Springer-Verlag, 2001.

[BS02] A. Berlea and H. Seidl. fxt A Transformation Language for XML Documents. *Journal of Computing and Information Technology*, 10(1):19–35, 2002.

[CD00] J. Cheesman and J. Daniels. *UML Components: a simple process for specifying component-based software.* Addison-Wesley, 2000.

[CG90] Nicholas Carriero and David Gelernter. *How to write parallel programs: a first course.* MIT Press, Cambridge, MA, USA, 1990.

[Cha85] Daniel Marcos Chapiro. *Globally-asynchronous locally-synchronous systems (performance, reliability, digital).* PhD thesis, 1985.

[Che76] Peter Pin-Shan Chen. The entity-relationship model – toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[Chu41] A. Church. *The Calculi of Lambda Conversion.* Princeton University Press, Princeton, N.J., 1941.

[Cla] J. Clark. XSL Transformations (XSLT) Version 1.0, W3C Recommendation 16 Nov 1999.

[Cla01] J. Clark. The Design of RELAX NG. Available at http://-www.thaiopensource.com/relaxng/design.html, 2001.

[com] The Rule Markup Initiative community.

[dBB00] Frank S. de Boer and Marcello M. Bonsangue. A Compositional Model for Confluent Dynamic Data-Flow Networks. In *MFCS '00: Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science*, pages 212–221, London, UK, 2000. Springer-Verlag.

[dBBJ$^+$04] Frank S. de Boer, Marcello M. Bonsangue, Joost Jacob, Andries Stam, and Leendert W. N. van der Torre. A Logical Viewpoint on Architectures. In *EDOC*, pages 73–83. IEEE Computer Society, 2004.

[dBBJ$^+$05] Frank S. de Boer, Marcello M. Bonsangue, Joost Jacob, Andries Stam, and Leendert W. N. van der Torre. Enterprise Architecture Analysis with XML. In *HICSS*. IEEE Computer Society, 2005.

[DH01] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[DJPV03] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Proceedings of the 1st Symposium on Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *LNCS Tutorials*, pages 70–98, 2003.

[DW98] Desmond D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach.* Addison-Wesley, 1998.

[ea04] H. Jonkers et al. Concepts for modeling enterprise architectures. In *International Journal of Cooperative Information Systems*, 2004.

[ea05] Marc Lankhorst et al. *Enterprise Architecture at Work: Modelling, Communication and Analysis.* Springer, December 2005.

[EFA90] S. Hupfer E. Freeman and K. Arnold. *JavaSpaces TM Principles, Patterns, and Practice.* Addison-Wesley, September 1990.

[EG94] G. Engels and L.P.J. Groenewegen. Specification of Coordinated Behavior by SOCCA. In B. Warboys, editor, *Proc. of the 3rd European Workshop on Software Process Technology (EWSPT '94)*, pages 128–151, Berlin, Germany, February 1994. Springer-Verlag.

[EJL$^+$99] Henk Eertink, Wil Janssen, Paul Oude Luttighuis, Wouter B. Teeuw, and Chris A. Vissers. A business process design language. In *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I*, pages 76–95, London, UK, 1999. Springer-Verlag.

[Fox88] G. C. Fox. Domain decomposition in distributed and shared memory environments I: uniform decomposition and performance analysis for the NCUBE and JPL Mark IIIfp hypercubes. In *Proceedings of the 1st International Conference on Supercomputing*, pages 1042–1073, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[GBR99] I. Jacobson G. Booch and J. Rumbaugh. *The Unified Modeling Language Reference Manual.* Addison Wesley, 1999.

[GCK02] David Garlan, Shang-Wen Cheng, and Andrew J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. *Sci. Comput. Program.*, 44(1):23–49, 2002.

[GMW97] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language, 1997.

[GS79] C. Gane and T. Sarson. *Structured Systems Analysis: Tools and Techniques.* Prentice Hall, Englewood Cliffs, 1979.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. pages 477–498, 1985.

[Jaca] Joost Jacob. The ASCII Markup Language (AML) white-paper. Available at http://homepages.cwi.nl/∼jacob/aml.

[Jacb] Joost Jacob. Coordinating UML with UnCL web site. Available at http://homepages.cwi.nl/∼jacob/uncl.

[Jacc] Joost Jacob. The OMEGA Component Model documents and implementation. Available at http://homepages.cwi.nl/∼jacob/component.html.

[Jacd] Joost Jacob. UML Kernel Model Semantics Demonstration. Available at http://homepages.cwi.nl/∼jacob/km/cgikm.html.

[Jac04a] Joost Jacob. A Rule Markup Language and Its Application to UML. In Margarita and Steffen [MS06], pages 26–41.

[Jac04b] Joost Jacob. The OMEGA Component Model. *Electr. Notes Theor. Comput. Sci.*, 101:25–49, 2004.

[JvBA$^+$03] H. Jonkers, R. van Buuren, F. Arbab, F.S. de Boer, M.M. Bonsangue, H. Bosma, H. ter Doest, L. Groenewegen, J. Guillen-Scholten, S. Hoppenbrouwers, M. Iacob, W. Janssen, M. Lankhorst, D. van Leeuwen, E. Proper, A. Stam, L. van der Torre, and G. Veldhuijzen van Zanten. Towards a language for coherent enterprise architecture description. In M. Steen and

B.R. Bryant, editors, *Proceedings 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003)*, pages 28–39. IEEE Computer Society Press, 2003.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[KMM00] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, June 2000.

[LKA$^+$95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Trans. Software Eng.*, 21(4):336–355, 1995.

[LMM99] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, page 465, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[MAS$^+$03] James McGovern, Scott W. Ambler, Michael E. Stevens, James Linn, Vikas Sharan, and Elias K. Jo. *A Practical Guide to Enterprise Architecture.* Prentice Hall PTR, 2003.

[Mat03] Mathematical Markup Language (MathML) version 2.0. Online, 10 2003.

[MOF] The Meta-Object Facility. Available at http://www.omg.org/-technology/documents/formal/mof.htm.

[MORW04] Michael Möller, Ernst-Rüdiger Olderog, Holger Rasch, and Heike Wehrheim. Linking CSP-OZ with UML and Java: A Case Study. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *IFM*, volume 2999 of *Lecture Notes in Computer Science*, pages 267–286. Springer, 2004.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, Berlin, January 1992.

[MRRR02] N. Medvidovic, D. Rosenblum, D. Redmiles, and J. Robbins. Modeling Software Architecture in the UML. *ACM Transactions on Software Engineering and Methodology*, 11(1):2–57, 2002.

[MS06] Tiziana Margarita and Bernhard Steffen, editors. *Leveraging Applications of Formal Methods, First International Symposium, ISoLA 2004, Paphos, Cyprus, October 30 - November 2, 2004, Revised Selected Papers*, volume 4313 of *Lecture Notes in Computer Science*. Springer, 2006.

[OME] OMEGA IST-2001-33522 Correct Development of Real-Time Embedded Systems. Website at http://www-omega.imag.fr/-index.php.

[OMG] The Object Management Group (OMG). Website at http://-www.omg.org/.

[OMG03] UML 2.0: Superstructure Specification, Final Adopted Specification ptc/03-08-03, 2003.

[ORR$^+$96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[Pau94] Lawrence C. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828:xvii + 321, 1994.

[Pet94] M. Pettersson. RML - A New Language and Implementation for Natural Semantics. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming, PLILP*, volume 884 of *LNCS*, pages 117–131. Springer-Verlag, 1994.

[PL99] Ivan Paltor and Johan Lilius. Formalising UML State Machines for Model Checking. In Robert B. France and Bernhard Rumpe, editors, *UML*, volume 1723 of *Lecture Notes in Computer Science*, pages 430–445. Springer, 1999.

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[pUM] The Precise UML Group. Available at http://www.cs.york.ac.uk/pumldt.

[RUL] The Rule Markup Initiative Community. Website at http://www.ruleml.org.

[SAdBB02] Juan Guillen Scholten, Farhad Arbab, Frank S. de Boer, and Marcello M. Bonsangue. Mocha: A middleware based on mobile channels. In *COMPSAC '02: Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, pages 667–673, Washington, DC, USA, 2002. IEEE Computer Society.

[SAdBB03] Juan Guillen Scholten, Farhad Arbab, Frank S. de Boer, and Marcello M. Bonsangue. A channel-based coordination model for components. *Electr. Notes Theor. Comput. Sci.*, 68(3), 2003.

[Sel98] Bran Selic. Using UML for Modeling Complex Real-Time Systems. In *LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 250–260, London, UK, 1998. Springer-Verlag.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-time object-oriented modeling.* John Wiley & Sons, Inc., New York, NY, USA, 1994.

[SJdB$^+$04] Andries Stam, Joost Jacob, Frank S. de Boer, Marcello M. Bonsangue, and Leendert W. N. van der Torre. Using XML Transformations for Enterprise Architectures. In Margarita and Steffen [MS06], pages 42–56.

[Soc00] IEEE Computer Society. IEEE std 1471-2000: IEEE recommended practice for architectural description of software-intensive systems, Oct. 9, 2000.

[SP99] Perdita Stevens and Rob Pooley. *Using UML: Software Engineering with Objects and Components*. Addison-Wesley, 1999.

[SRMM00] G. Schadow, D. Russler, C. Mead, and C. McDonald. Integrating medical information and knowledge in the HL7 RIM. *Proc AMIA Symp*, pages 764–8, 2000.

[SS99] Steve Schneider and S. A. Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[SWB03] Perdita Stevens, Jon Whittle, and Grady Booch, editors. *UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings*, volume 2863 of *Lecture Notes in Computer Science*. Springer, 2003.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[U20] The U2 Partners. Available at http://www.u2-partners.org.

[vdB01] Michael von der Beeck. Formalization of UML-Statecharts. In *UML '01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 406–421, London, UK, 2001. Springer-Verlag.

[vR95] G. van Rossum. The Python Reference Manual. Technical Report ISSN 0169-118X, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1995.

[W3C] The World Wide Web Consortium (W3C). Website at http://www.w3c.org.

[Wil05] Gregory V. Wilson. Extensible programming for the 21st century. *Queue*, 2(9):48–57, 2005.

[XMI] The XML Metadata Interchange Format (XMI). Available at http://www.omg.org/technology/documents/formal/xmi.htm.

[XML] The Extensible Markup Language (XML). Available at http://www.w3.org/XML/.

[XMS] The XML Schemas. Available at http://www.w3.org/XML/-Schema.

[XSL] The Extensible Stylesheet Language Family (XSL). Available at http://www.w3.org/Style/XSL/.

[Zac87] John A. Zachman. A Framework for Information Systems Architecture. *IBM Syst. J.*, 26(3):276–292, 1987.

Summary

It is desirable to model software systems in such a way that analysis of the systems, and tool development for such analysis, is readily possible and feasible in the context of large scientific research projects. This thesis emphasizes the methodology that serves as a basis for such developments. I focus on methods for the design of data-languages and their corresponding tools. A recurring problem in large software research projects is that even though every partner uses their own version of such languages and tools, the semantic consistency of these different versions still has to be proven. This so–called consistency problem is a pivotal theme in this thesis. A second theme consists of another problem, the so–called adaptation problem, where existing modeling languages are being used to develop a new semantic basis, for instance for visualization and simulation techniques. For this second problem the contribution of this thesis consists of the development of tools for automatic transformation of data–languages; this is how the research for this thesis could contribute to the projects Omega and Archimate that sponsored the research.

As an example the sieve of Eratosthenes is modeled in such a way that the model is as abstract as possible, but still being consistent with the so–called Kernel–model semantics in the Omega project. The consistency is derived from the corresponding tool being executable, such that it yields the desired observable behavior. A second contribution is visualization and the architecture of tools for visualization and simulation. These are developed for business processes, so–called Enterprise Architectures, in the Archimate project, where their architecture is outlined. For this purpose a so–called domain–specific data language is introduced in order to be able to model large software projects. This is further described in the Introduction, chapter 1. The other chapters consist of my publications for these projects.

Samenvatting

In de context van grote wetenschappelijke onderzoeksprojecten in de Informatica is het wenselijk software systemen zodaning te modelleren dat analyse van deze systemen goed mogelijk is. Ook analyse ondersteunende tool–ontwikkeling dient mogelijk te zijn. In dit kader leg ik in dit proefschrift de nadruk op de methodiek die hieraan ten grondslag ligt. Hiertoe richt ik me in het bijzonder op methoden voor het ontwerp van data–talen en van de bijbehorende tools. Het probleem hierbij is dat in grotere software onderzoeksprojecten weliswaar elke partner zijn eigen versie van deze talen en tools gebruikt, maar dat dan de consistency van de bij deze versies behorende semantieken nog bewezen moet worden. Dit probleem staat in het proefschrift centraal als het zogenaamde consistency problem. Een tweede probleem is het zogenaamde adaptation problem, waarbij van bestaande modelleringstalen uitgegaan wordt met het doel een nieuwe semantische basis, bij voorbeeld voor visualisatie- en simulatie–technieken, te ontwikkelen. Voor dit laatste bestaat de bijdrage van dit proefschrift uit de ontwikkeling van tools voor de automatische transformatie van data–talen en hun bijbehorende software; op deze wijze kon het onderzoek voor dit proefschrift bijdragen aan de projecten Omega en Archimate die het onderzoek gefinancierd hebben.

Als voorbeeld wordt de zeef van Eratosthenes gemodelleerd op een zo abstract mogelijke wijze, die consistent is met de zogenaamde Kernel–model semantiek van het Omega project. Deze consistentie wordt afgeleid uit de executeerbaarheid van de bijbehorende tool die hiervoor ontwikkeld is, aangezien deze het gewenste observeerbare gedrag oplevert. Een tweede bijdrage wordt geleverd op de gebieden van visualisatie en de architectuur van tools voor visualisatie en simulatie. Als voorbeeld zijn dergelijke tools ontwikkeld voor bedrijfsprocessen, zogenaamde Enterprise Architectures, in het kader van het Archimate project, en wordt hun architectuur geschetst. Hiertoe wordt een zogenaamde ”domain–specific data language” voorgesteld teneinde de vakdiscussie in bestaande grote software projecten te kunnen modelleren. Dit wordt in de Introduction, hoofdstuk 1, uitgewerkt. De resterende hoofdstukken bestaan uit mijn publicaties voor deze projecten.

Curriculum Vitae

Joost Jacob werd geboren op 5 februari in Haarlem. Hij bracht zijn jeugd door in het Gooi, waar hij met goed gevolg het VWO-diploma behaalde aan het Sint Vituscollege in Bussum.

In 1981 ging Joost scheikunde studeren in Leiden. Tijdens de studie scheikunde deed Joost een bijvak informatica dat hem aansprak. Joost ging het geleerde op informatica gebied toepassen bij verschillende bedrijven, vooral bij automatisering in de juridische branche. Dit bleef zo tot 2002, als zelfstandig ondernemer in Leiden.

In 1997 begon Joost met de avondstudie informatica aan de Universiteit Leiden, en werd doctorandus informatica in 2002. Van 2002 tot 2006 werkte Joost op het Centrum voor Wiskunde en Informatica (CWI) in Amsterdam, in de thema–groep SEN3 van prof. dr. Jan Rutten. Het werk bij het CWI leidde tot verschillende publicaties in internationale conferenties en workshops die in dit proefschrift terug te vinden zijn. De projecten waarin Joost werkte waren het internationale project Omega, door de Europese Commissie gesponsord, op het gebied van real–time embedded systems, het nationale project ArchiMate over bedrijfsarchitectuur, en het bilaterale NWO en DFG project Mobi–J over "assertional" methoden voor mobiele asynchrone kanalen in Java. Na 2006 heeft Joost bij verschillende bedrijven in de ICT gewerkt.

Joost woont in Leiden met zijn Monique en zijn twee zonen Pepijn en Marnix.