MDD4SOA
Model-Driven Development for Service-Oriented Architectures

Philip Mayer

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig–Maximilians–Universität München

zur Erlangung des Grades
Doctor rerum naturalium (Dr. rer. nat.)

vorgelegt von
Philip Mayer

München, 22. Oktober 2010

Erstgutachter	Prof. Dr. Martin Wirsing Ludwig-Maximilians-Universität München Deutschland
Zweitgutachter	Dr. Stephen Gilmore The University of Edinburgh Scotland
Tag des Rigorosums	23. November 2010

Abstract

This thesis presents a methodological as well as tool-based integration of Model-Driven Development (MDD) and Service-Oriented Computing (SOC). Model-driven development is a process for creating software by employing models to design and analyse a system before creating source code, while service-oriented computing is based on Service-Oriented Architectures (SOAs) for structuring software systems by using the concept of services, which are self-describing networked software artefacts offering a particular encapsulated functionality.

Four main results are contributed to this area. First, we provide a UML-based, domain-specific modelling language specifically targeted at modelling service behaviour and service protocols. Secondly, these models are given a rigorous semantics based on transition systems, and a verification method is provided. Thirdly, we introduce model transformations to be used to generate code from UML models of SOA software systems for the Web Service standards platform as well as a traditional object-oriented language. Fourth, we provide a testing environment for the generated code, and relate the code generation and formal semantics areas with a simulation and tracing approach.

Model-driven development of service-oriented architectures (MDD4SOA) consists of many individual steps and shows its full potential when automated. Each of the contributions of this thesis is thus fully tool-supported, providing modelling assistance, automated analysis, and generation of executable source code. Furthermore, a common development environment is provided in which MDD4SOA-related tools can be integrated and combined in workflows as required by the development task at hand.

Taken together, the methods and tools provided in this thesis form the blueprint of an UML-based, analysis-supported, model-driven development approach to service-oriented software.

Zusammenfassung

Diese Arbeit stellt einen Ansatz zur methodologischen sowie werkzeugunterstützten Integration modellgetriebener Entwicklung (MDD) und dienstorientierter Datenverarbeitung (SOC) vor. Die modellgetriebene Entwicklung ist ein Prozess zur Herstellung von Software, bei welchem vor Erzeugung des Quelltexts Modelle zum Entwurf und zur Analyse eines Systems eingesetzt werden; dienstorientierte Datenverarbeitung nutzt dienstorientierte Architekturen (SOAs) zur Strukturierung von Softwaresystemen mittels des Dienst-Konzepts. Dienste sind selbstbeschreibende, vernetzte Softwareartefakte, welche spezifische gekapselte Funktionalität bereitstellen.

Vier wichtige Ergebnisse werden zu diesem Gebiet beigesteuert. Zunächst wird eine UML-basierte, domänenspezifische Modellierungssprache bereitgestellt, welche genau auf die Bedürfnisse der Modellierung von Dienstverhalten und Dienstprotokollen zugeschnitten ist. Zweitens erhalten so erstellte Modelle eine rigorose Semantik auf der Basis von Transitionssystemen; außerdem wird eine Verifikationsmethode bereitgestellt. Drittens werden Modelltransformationen zur Generierung von Quelltext aus UML-Modellen eingeführt — sowohl mit dem Ziel der Web Service Standards Platform als auch einer traditionellen objektorientierten Sprache. Viertens wird eine Testumgebung für den generierten Quelltext bereitgestellt, und durch einen Simulations- und Nachverfolungsansatz wird eine Beziehung zwischen Quelltexterzeugung und formaler Semantik hergestellt.

Die modellgetriebene Entwicklung dienstorientierter Architekturen (MDD4-SOA) besteht aus vielen einzelnen Schritten und zeigt ihr ganzes Potential erst durch Automatisierung. Jeder der einzelnen Beiträge dieser Arbeit ist daher vollständig werkzeugunterstützt, wodurch Hilfestellung bei der Modellierung, automatisierte Analyse, und die Erzeugung ausführbaren Quelltexts ermöglicht wird. Darüber hinaus wird eine gemeinsame Entwicklungsumgebung bereitgestellt, in welche die MDD4SOA-Werkzeuge integriert und wie von der jeweiligen Entwicklungsaufgabe verlangt zu Arbeitsabläufen zusammengefasst werden können.

Zusammengefasst bieten die Methoden und Werkzeuge in dieser Arbeit die Grundlage für einen UML-basierten, durch Analysemethoden unterstützten, modellgetriebenen Softwareentwicklungansatz für dienstorientierte Software.

Acknowledgements

Firstly, I would like to thank my supervisor Martin Wirsing for his support during my work at the chair for Programming and Software Engineering (PST), both with regard to my thesis and other projects I've been involved in. I am very grateful for his friendliness and respect which I will remember always.

Staying at PST, I would like to extend my thanks to all of my (former) colleagues and will always remember the good times we had both in Munich and on various mountains in the Bavarian alps. Most especially, I'd like to thank Andreas Schroeder, my room mate and fellow coder, who has been a great source of inspiration for many of the topics found in this thesis. I am also very grateful for the help given by Sebastian Bauer and Nora Koch for their support regarding MIOs and UML, respectively. Matthias Hölzl has greatly helped in improving the text of this thesis.

The bulk of my research has been done in the context of the EU project Sensoria. Here, I'd particularly like to thank Howard Foster and Mirco Tribastone for their support of my work and the good times we had at the Sensoria meetings. This also holds true for Stephen Gilmore, who I am additionally very grateful to have as my second reviewer and whose helpful suggestions have greatly improved my thesis.

Moving to a new city is never easy, and settling in requires open-hearted people to relate to. I am very lucky to have found such people. My thanks go out to the entire "Monaco Gang", but most of all to my former colleague Florian Lasinger for welcoming me to Munich and feeling right at home.

And last, but certainly not least — special thanks go to Stephanie Fichtner for her love and support through all these years!

Contents

Chapter 1

Introduction

Software based on Service-Oriented Architectures (SOAs) [Erl05, SD05] consists of individual, self-describing networked components called *services*, which each implement and offer a certain encapsulated set of functionality. Central registries provide an index of available services to be discovered and used by the participants of the system, which communicate with each other via standardised protocols. Services can be combined to yield new services, thus enabling compositional development of software systems.

Since their inception, SOAs have quickly gained support in both industry and academia as the main architectural paradigm for developing enterprise software applications. However, the promised benefits of SOAs — such as high interoperability, maintainability, and the ability to evolve in small steps — come at the price of increased complexity of SOA artefacts and thus development of SOA systems. This has led to the unsatisfying situation that SOA development today is mostly done in an ad-hoc way: Software development processes, methods, and tools have not yet caught up with the new requirements of the SOA paradigm.

For example, the Unified Modeling Language (UML) as one of the main graphical languages for modelling software systems does not contain any specific constructs for SOA systems; profiles which add this support are only slowly emerging. Furthermore, theoretical foundations for systems build on SOAs are not yet integrated with high-level modelling approaches, which is required to validate system design and instill trust in SOA implementations. Lastly, SOA design has a lower granularity than traditional software; it represents a new abstraction step away from computer hardware towards software artefacts understandable and directly relevant on a business level. Ad-hoc industry approaches for creating SOA systems with existing technology such as object-oriented languages fall short of fully embracing, and exploiting, this abstraction step.

As the use of SOAs now starts to permeate the software industry and is becoming a requirement for the design of many large software systems, employing unified, SOA-aware software development methods, tools, and processes is of paramount importance to be able to deal with the complexity, but also with the opportunities inherent to the SOA paradigm.

Model-Driven Development (MDD) [Sel03] is a software development process which enables the evolution of existing methods, tools, and processes towards a more structured, rigorous approach to the development of SOA systems [Fra03]. MDD focuses on *models* of a software system, which represent an abstract view of the system to be developed. The modelling principle is well-known in traditional engineering methods. Transferred to software engineering, the benefit of using models is in fact not restricted to communication of the design, but promises to be even greater: Through automation, models can drive the system implementation or even replace the need to implement a system by hand.

Another benefit of models is enabling early analysis of software systems. Using formal methods, a rigorous semantics can be designed which is amenable to verification and thus yields results for improving the software system.

This thesis shows how to integrate the SOA and MDD concepts, and presents a unified approach to *model-driven development of service-oriented systems* (MDD4SOA). The MDD4SOA approach is comprehensive as it provides a semantic foundation, formal analysis, and tool support throughout the development.

An introduction to MDD4SOA is given in section 1.1. Section 1.2 gives a first look at the contribution of this thesis. MDD4SOA has been developed within the Sensoria EU project, which we reference in section 1.3. For discussing the individual results of this thesis, we employ a case study from the domain of computer-assisted university management, which is introduced in section 1.4 and elaborated in further chapters. Finally, the structure of this thesis is shown in section 1.5.

1.1 MDD4SOA

This thesis develops a complete end-to-end approach to model-driven development for service-oriented systems, starting with SOA models created in an appropriate modelling language, performing formal analysis on these models to provide early feedback about design problems, and finally using the SOA models to generate code in executable target languages, which can also be validated against the formal model. All areas are based on a meta-model for services, and are complemented by a common tooling platform which allows the use of tools supporting these areas as well as externally contributed tools in an integrated development environment.

A graphical overview of the different contribution areas of the MDD4SOA approach is shown in figure 1.1. The individual areas are discussed in the following sections.

1.1.1 Modelling

The first area of the MDD4SOA approach shown in figure 1.1 and thus the first part of this thesis is concerned with modelling SOA artefacts. We have chosen the Unified Modeling Language (UML) [OMG10b] as the basis for this step.

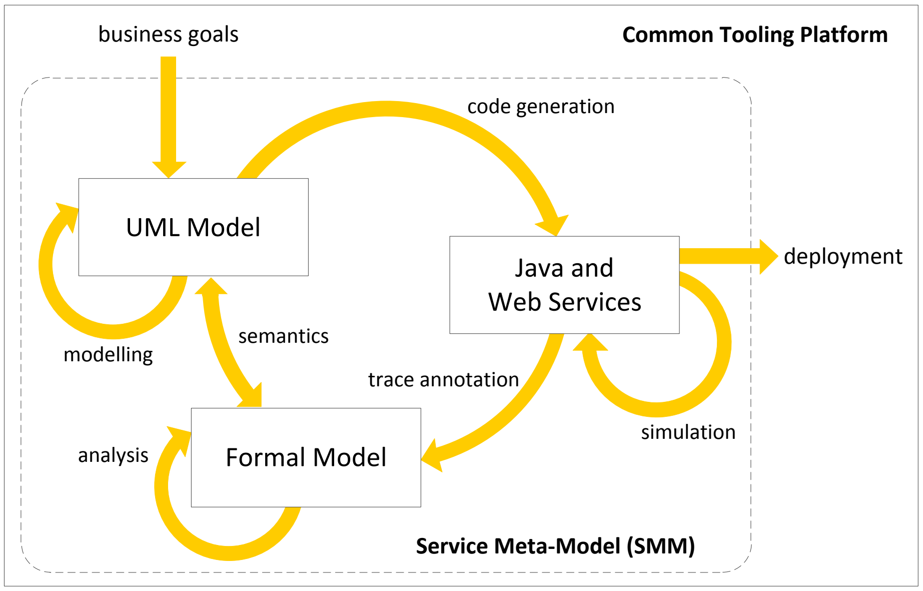


Figure 1.1: The MDD4SOA Approach

UML is a well-known and mature language for software design, and is regarded as the de facto standard for software modelling. However, as we will point out in the corresponding chapter, UML lacks specific elements for modelling services (and in particular, service behaviour). This thesis thus contributes adequate support to the UML in the form of a profile, which is called UML4SOA.

Modelling support for SOAs is particularly promising as SOA applications tend to implement complex business functionality. Furthermore, services are usually of lower granularity than traditional components. Thus, a more abstract view comes naturally to a service-based architecture in comparison to traditional programming languages.

1.1.2 Service Meta-Model (SMM)

In order to be able to discuss formal analysis and code generation for UML-4SOA, this thesis defines a Service Meta-Model (the SMM), which offers the necessary meta-classes for describing complete SOA systems. This includes static, dynamic, and data handling aspects. The SMM is shown as an enclosing, dashed rectangle in figure 1.1.

The UML4SOA profile already mentioned above, along with parts of the UML, forms the concrete syntax of the SMM. The next two areas of MDD4SOA further build on the SMM, first defining a formal semantics with analysis support and, secondly, code generation.

1.1.3 Semantics and Analysis

One of the main benefits of using a model-driven development approach is the availability of models early in the development phase. Depending on the current level of abstraction, the models may or may not be complete, but should be correct in the sense that they are an appropriate representation of the final system. This opens up the possibility of model analysis.

Formal verification requires the introduction of a rigorous semantics for a model specification on which analysis can be performed. This thesis provides such a semantics for UML4SOA (via the Service Meta-Model) which is based on *modal input/output transition systems* [LNW07a], a specialised form of transition systems [And94]. Furthermore, we provide an analysis method for checking these automata for protocol compliance using *interface theories* [BMSH10], which, in turn, allows verification of UML4SOA models.

1.1.4 Code Generation

It has been remarked [Sel03] that models of software systems offer another benefit which is not available in traditional, non-software engineering approaches: The ability to automatically or semi-automatically *generate* the resulting system from a model. Due to this possibility, MDD has also been described as the next step in programming language evolution: Instead of generating byte code or machine code, an *MDD compiler* will instead generate code in traditional programming languages such as Java from higher-level models of the system [MCF03].

This thesis has addressed the generation of executable code from UML4SOA (again via the Service Meta-Model) by investigating two *model transformations*: The first uses Web Services [WCL$^+$05] as the target platform, while the other employs a traditional programming language (Java) [GJSB05]. The use of different execution platforms illustrates another benefit of MDD, which is the ability to switch concrete execution platforms by adapting the transformers.

1.1.5 Simulation and Tracing

In order to be able to execute the generated code and observe its behaviour, we use a simulation and tracing approach. Through an automated testing environment, the generated code is executed, creating a trace of occurring events (simulation). Furthermore, this trace can be compared with the transition systems semantics, yielding information about how the paths through the automata are used during runtime (trace annotation). This approach furthermore allows validation of the generated code.

1.1.6 Tooling

Each step in the MDD4SOA approach and thus *each internal arrow* in figure 1.1 is tool-supported; an overview of all tools is shown in section 8.3 (page 287).

However, we believe that besides the availability of tools for the individual steps shown above, a coherent development platform which integrates these tools (and related ones) is of great importance to foster the use of model-driven development techniques for service-oriented software.

Thus, a common tooling platform has been developed as part of this thesis; it is shown as the encompassing element of figure 1.1. In fact, the service-oriented principle has been applied to this tooling platform as well: Each tool is integrated as a service, and may be combined with other tools to form a workflow as required by the development task at hand.

1.2 Contributions of this Thesis

The contribution of this thesis is the investigation of and research into a full end-to-end approach to model-driven development of service-oriented architectures, which includes modelling, analysis, code generation, and automated testing, along with an integrated tooling environment.

- The contribution in the *modelling* domain lies in the introduction of a lightweight, non-intrusive extension for service behaviour to the UML. As pointed out in the corresponding chapter, the UML4SOA profile achieves this goal with a minimum amount of changes to the UML itself. UML-4SOA is already based on the SoaML OMG standard [OMG09b] which is currently in beta status, and thus is readily usable as the behavioural counterpart of SoaML by UML modellers.

- With regard to *semantics and analysis*, the contribution lies in the definition of a rigorous semantics for UML4SOA (via the Service Meta-Model) and the introduction of a domain-specific interface theory for early protocol verification of SOA models. Another important aspect in this regard is the availability of full tool support for formal verification, which includes the automated generation of the formal model. Furthermore, the tools do not require developers to understand or even see the formal backing.

- In the area of *code generation*, we contribute two model transformations which enable generation of executable code from UML4SOA (again, via the Service Meta-Model) for two industry-standard platforms. The key contribution here lies in the transformation descriptions which show how the mapping of high-level UML4SOA models to actual code is carried out.

- The *simulation and tracing* approach contributes the ability to directly execute the generated implementations, thus completing the MDD approach and furthermore enabling a comparison of the behaviour of the generated implementations against the formal semantics of UML4SOA. This part also contributes tools for automating the simulation and annnotation process.

- Finally, the contribution of the *integrated tooling platform* lies in the successful integration of different kinds of tools through a lightweight, SOA-based architecture. We believe that due to its open structure, this approach can serve as a blueprint for development tool integration.

It is important to note that the contributions of this thesis do not only lie in the individual methods and tools listed above, but also in their integration. By taking part in a fully automated and integrated process from modelling via analysis to code, the individual steps benefit from each other, improving and streamlining the MDD approach as a whole.

As an example for this integration, we use a case study which is first introduced in section 1.4. In each subsequent chapter, this case study is used to illustrate the results and benefits of the corresponding methods and tools; thus, the case study provides an end-to-end example of the development of a SOA system in a model-driven way.

1.3 Sensoria

MDD4SOA has been developed while the author took part in Sensoria, an integrated project funded by the European Commission (EC) which took place from 2005 to 2010. For a comprehensive overview of Sensoria, the reader is referred to [WBC+09].

The aim of the Sensoria project has been the development of a novel comprehensive approach to the development of service-oriented overlay computers, and as such, some of the results presented in this thesis have contributed to the core of Sensoria. In particular, the following three main research objectives of Sensoria have a direct relationship with this thesis:

- *Service-Oriented Extensions of the UML.* This objective addresses the need for primitives for modelling SOA systems in UML. UML4SOA has been developed as an answer to this objective.

- *Model Transformations.* Sensoria also addresses the need for model transformations, both for generating code and for mediating between different formal tools. The model transformers and code emitters provided as part of this thesis are a contribution to this objective.

- *Tool Integration.* Finally, Sensoria addresses tool support and in particular tool integration as a major requirement for the development of SOA systems. The common tooling platform introduced in this thesis has been developed to meet this objective.

Each of the methods and tools introduced in this thesis have been applied to the case studies provided by the Sensoria project, which has led to important feedback from both industry and academia for improving the results of this thesis.

1.4 The eUniversity Case Study

For discussing the results of this thesis, we employ a common case study throughout all chapters. The case study is taken from the domain of computer-assisted university management

The administration of a university is a complicated task. Student applications, enrolment, course management, theses, and examination management all pose individual problems and, in general, a lot of paperwork. Nowadays, many of these tasks can be and are being automated. As universities are often large organisations with autonomous sub-organisations, a promising approach for this is the use of SOA-based software, in which the individual parts of a university as well as (external) students can work together with respective back- and front-ends of a web-based system.

To investigate the problem of developing SOA-based university management systems, the Sensoria project includes a case study based on a set of university scenarios that make use of the specific features of SOAs [Hö7]. In particular, we consider eUniversities, i.e., universities in which at least all of the paperwork, if not the courses themselves, are handled online.

The chosen scenario for this thesis is the problem of *Thesis Management*. In this scenario, we have considered the management of a student thesis (bachelor, master, or diploma) from the initial announcement to the final grading.

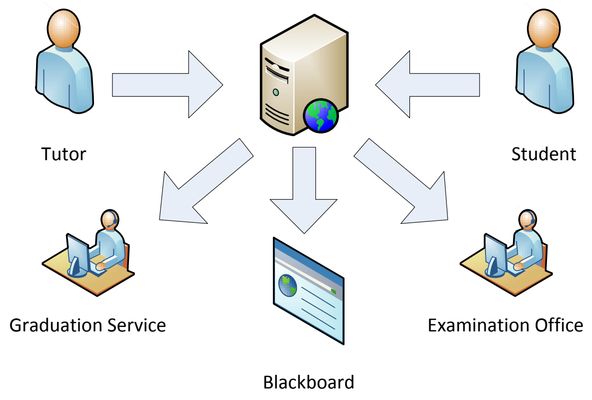


Figure 1.2: eUniversity Case Study: Overview

The scenario consists of six partners and computing systems working together. As shown in figure 1.2, a tutor (left) first provides a new thesis proposal — bachelor, master, or diploma — to a central server (middle), which distributes it to a university-wide blackboard (bottom centre) to inform students of this opportunity. Once a student (right) decides to pick up a thesis, the central server starts the thesis, informing both the examination office (bottom right), and registering the student for the graduation ceremony (bottom left).

While the thesis is in progress, the student may provide updates, which the tutor may retrieve. Once the thesis is declared to be finished, an assessment is requested from the tutor. If the assessment is positive, the examination office is instructed to issue the corresponding certificates. If not, the examination office is informed of the problem, and the graduation service needs to unregister the student from the graduation ceremony.

The thesis management scenario is an interesting case study as it integrates several autonomous organisational entities which each provide their own services, as well as two clients of the system with different interests and required functionality. Thus, it provides us with the complex multiparty interactions typical of systems based on service-oriented architectures.

In each of the following chapters, this case study is revisited to illustrate newly introduced concepts and their effect on the case study model.

1.5 Thesis Structure

The structure of this thesis follows the development process laid out in figure 1.1. We begin with introducing the relevant background information which sets the stage for later chapters in chapter 2. This includes an overview of Service-Oriented Architectures (SOAs), Model-Driven Development (MDD), parts of the Unified Modelling Language (UML), the UML profile SoaML, the domain of modal input/output automata and interface theories, as well as an overview of technical resources used in this thesis.

Chapters 3, 5, 6 and 7 introduce the main contributions of this thesis; they are grouped around the discussion of the Service Meta-Model (SMM) in chapter 4. Chapter 3 introduces a profile for modelling service behaviour in UML, and how this behaviour may be attached to the artefacts provided by the SoaML profile. The formal semantics and analysis methods are discussed in chapter 5. The third area of MDD4SOA, model transformations and code generation with the target of the Web Services family and Java, are discussed in chapter 6. Finally, we discuss the simulation and tracing approach for the generated code in chapter 7.

Last, but not least, our tool integration platform — the Service Development Environment[1] — is introduced in chapter 8, along with a description of integrated tools, and tool chains which demonstrate the ability of combining tools using the platform.

A summary of the provided methods and tools of this thesis along with a review of achievements and an outlook for future work is given in chapter 9.

[1]Formerly named Sensoria Development Environment.

Chapter 2

Setting the Stage

This chapter introduces the underlying concepts and technologies which form the basis for the main chapters of this thesis.

The chapter begins with an introduction of the research areas to which the results of the thesis contributes. Section 2.1 introduces the area of service-oriented computing and the Web Services standards family as the current industry standard for realising SOAs. In section 2.2, a brief overview of the Unified Modeling Language (UML) is provided, along with an introduction into the extension mechanisms of the UML used later in the thesis.

Both the SOA and the UML sections are brought together in section 2.3 with the introduction of the UML profile SoaML, which captures the static part of SOA modelling in UML and is a prerequisite to the profile introduced in chapter 3.

The concept of model-driven development used in chapter 6 is introduced in section 2.4. Section 2.5 introduces the formal framework of modal input/output automata and interface theories used for formal verification.

All results presented in this thesis are supported by tools. For the implementation of these tools, a series of frameworks and libraries are used which are introduced in section 2.6.

2.1 Service-Oriented Computing

Service-Oriented Computing, or SOC for short, refers to a computing environment built on *services*, which are software artefacts providing a certain set of functionality, in most cases via a network such as the Internet. Enabling Service-Oriented Computing requires a certain structure to be followed in the implementation of software systems, which is referred to as the Service-Oriented Architecture (SOA).

While the term SOA only refers to an arbitrary software design which is based on services, concrete implementations and standards families have been created which allow a precise and technical specification of each component in a SOA and their collaboration. The most popular realisation of SOAs are Web Services, which are defined through various documents in the Web Service family of standards.

The following two subsections introduce the abstract concepts of SOAs, and the realisation of SOAs through Web Services, respectively.

2.1.1 Service-Oriented Architectures

The Service-Oriented Architecture (SOA) is an architectural paradigm which has gained great momentum in the industry in recent years. SOA-based systems are the latest approach to building, integrating, and maintaining complex enterprise software systems. Although it is hard to find an exact definition, a SOA is generally regarded as having the following basic characteristics ([WCL^{+}05], [ACKM03], [EN06]):

- The basic building blocks of a SOA are *services*, which are distributed and invoked via a network, and should be loosely coupled.
- Services are described in some sort of *abstract interface language*, and can be invoked without knowledge of the underlying implementation through standardised communication protocols.
- Services can be *dynamically discovered and used*; the discovery process is based on meta-data *about* the required services.
- A SOA supports integration, or *composition*, of services in a recursive way, i.e. compositions of services can again be regarded as services.

SOAs are closely tied to the hope of being able to re-structure the intra- and inter-enterprise software landscape to allow greater flexibility, thus being able to respond more quickly to changing business requirements.

Distributed architectures and integration methods have been around for some time, and SOA has evolved out of these methods rather than being a completely new concept. In fact, SOAs take intra-enterprise integration systems to a new level — as companies are outsourcing parts of their business or are cooperating with partners, their information systems also grow across company borders. Additionally, SOAs enable re-use of existing applications by wrapping them as services.

A very important feature of SOAs is the concept of service compositions. Environments which support service composition have their roots in workflow management, where business logic was implemented by composing existing, coarse-grained applications [ACKM03]. SOAs take composition to the next level by offering a much more homogeneous environment, as all parts of a composition are services themselves, (ideally) described in the same fashion, and communicating with the same messaging standards. One key application area of service composition is the realisation of business processes as a composition of services, thereby placing SOAs at the very heart of enterprise IT.

Service composition differs from traditional composition approaches in two important ways:

- Service composition is recursive. A composition of services is, in most cases, a service itself, which can be composed even further. This is an elegant way of dealing with complexity in large service-based systems.

- Service composition works in a distributed fashion. In traditional composition approaches, components are compiled, linked, included, and sold with the final composition. Service compositions, on the other hand, use existing services as-is, invoking them via a network.

There are two distinct ways for designing composite services, which have been named choreography and orchestration [SD05]:

- *Choreography.* A choreography describes a collaboration between services to achieve a certain goal. The control logic of a choreography is distributed — each service knows what to do and whom to contact; these actions are not described as part of the choreography. A choreography is an abstract definition of an interaction, intended to convey the general purpose and goal of the composition.

- *Orchestration.* An orchestration, on the other hand, focuses on one service, specifying the concrete actions to take to implement that service by using other services. The control logic is therefore centralised in the orchestration. An orchestration is intended to be executed.

From a more technical perspective, there are three key features of SOA-based systems which pose challenges, but also offer benefits for implementations:

- First, SOA-based systems are *open systems*: Invocation of service functionality should be possible without having access to the complete system; not only due to the distributed deployment of services but also due to their abstract (self-) descriptions and standardised protocols for interoperability.

- Second, a SOA consists of independent services, which talk to one another via a network. Thus, *communication* between services is a key aspect of SOAs. Although the complexity of network calls is still a challenge for system integration, the hope is to abstract from this layer when writing applications on a SOA level.

- Third, the distributed nature of SOAs render it very hard to guarantee atomic behaviour, i.e. an all-or-nothing semantics for certain tasks. Thus, another key concept of SOAs is support for *compensation*, which is used to explicitly state behaviour to roll-back, or *undo*, already successfully completed work.

Systems based on service-oriented architectures can be realised in a multitude of languages and can take very different technical representations — some are implemented on top of existing paradigms (such as in Java or the languages of the .NET platform); in others, new languages with a specific focus on SOAs have been created. The most successful family of standards for the implementations of SOAs are Web Services, which are detailed in the next section.

2.1.2 The Web Service Standards Family

Web Services are, arguable, the most important realisation of a SOA today. The Web Service architecture consists of over a dozen standards, which are managed by standardisation bodies like W3C [W3C10] and OASIS [OAS10][1]. The Web Service movement is backed by large industry players like IBM, Microsoft, Oracle, and others.

There are many different definitions of a Web Service. Even the W3C, which is involved in many of the basic Web Service standards, has two definitions available ([Wor04a], [Wor04b]). The first definition is relatively abstract:

> *A software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artefacts. A Web Service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols [Wor04b].*

A few things should be noted with regard to this definition. Firstly, the definition describes a Web service as a software application, which can basically mean anything from a small script to a large, server-spanning enterprise software system. In other words, the Web Services technology does not presume any specific size of Web Services; this lies in the responsibility of the interface designer. Secondly, a Web Service is self-describing, and must be discoverable, with the use of XML. The use of XML is important as it is a text-only, open standard, and if properly designed, XML documents may be read by humans and easily manipulated. Thirdly, a Web Service does not interact with humans, but with other software agents. Finally, as can be seen from this definition and in fact from the name Web Services itself, the Web plays a major role in the Web Service standards. The definition not only mentions the concept of a URI, but also Internet-based protocols as the message exchange layer.

W3C has another definition available, which already includes the names of some of the most important W3C Web Service standards, which are described below:

> *A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed*

[1]OASIS has also published a reference model for SOAs in [OAS06].

> *by its description using SOAP messages, typically conveyed using HTTP with an XML serialisation in conjunction with other Web-related standards [Wor04a].*

Web Services offer a significant step forward from existing middleware and EAI solutions due to three aspects, which are identified in [ACKM03]:

- *Service Orientation.* As a realisation of a SOA, service orientation comes natural to Web services. However, inherent loose coupling between the components of a system — each service is independently implemented — is a major change from older middleware systems.
- *Peer-to-Peer Middleware Protocols.* The runtime environment of Web Services — which consists of inter-enterprise space — offers no place for a central coordinator for management of resources and locks. Instead, the various peers must be able to agree upon such things on a bilateral basis.
- *Standardisation.* The aim of Web Services is to offer cross-enterprise integration, which necessitates industry-wide standards. The Web Service movement has been successful in many ways in introducing such standards by employing standardisation bodies and major industry players.

The first three standards of the Web Service technology stack are SOAP, WSDL, and UDDI, which have been published around the year 2000. SOAP (which is not an acronym) defines the basic messaging standard of Web Services; WSDL (Web Service Description Language) is an interface description language for Web Services. Finally, UDDI (University Discovery, Description, and Integration Service) handles dynamic discovery of Web Services. An overview of how SOAP, WSDL, and UDDI are related to one another and other important Web Service standards is shown in figure 2.1.

The Web Service architecture stack plays a major role as a transformation target in chapter 6. The following standards from the stack are relevant for this chapter:

- *BPEL.* The Business Process Execution Language (BPEL) is a high-level language for describing service and service orchestration behaviour. BPEL processes are based on many of the lower-level artefacts, and contain primitives for direct communication with other services without requiring lower-level networking details.
- *WSDL.* The Web Service Description Language (WSDL) is used to declare the functional interfaces of Web Services (which includes BPEL processes). Using the notions of services, ports, and messages, WSDL allows the specification of available services, operations supported by a service, and the data structures expected to arrive and to be sent out.
- *SOAP.* SOAP defines how messages to be sent or retrieved between services are structured and encoded. A SOAP definition for each provided

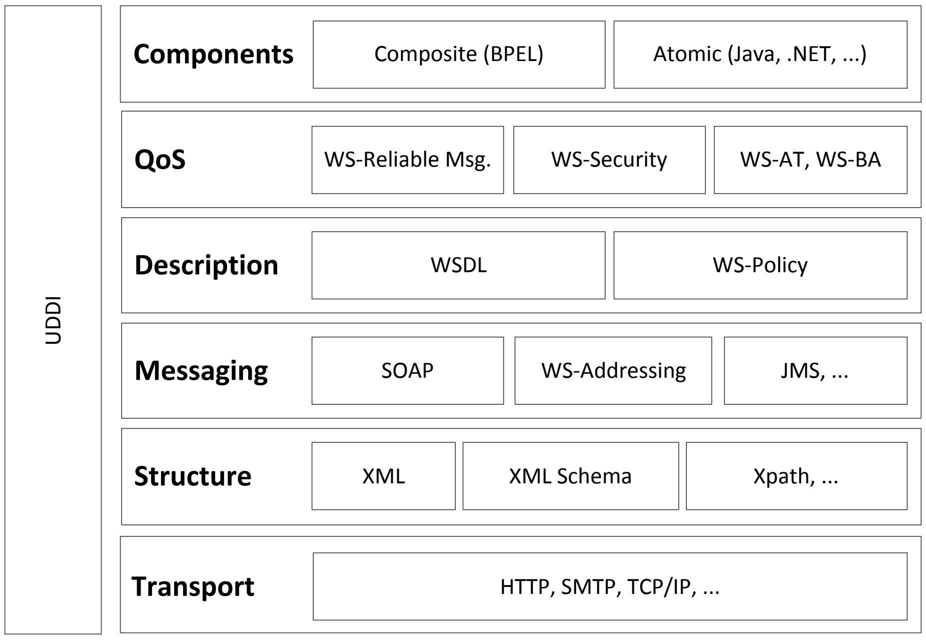


Figure 2.1: Web Service Architecture Stack

and required service is thus a necessary requirement for a successful invocation.

- *WS-Addressing.* In order to allow asynchronous callbacks to partners which are previously unknown, the location of a caller must be part of or attached to a message. The WS-Addressing standard provides the ability for such annotations.

- *XML Schema.* Both BPEL programs and WSDL definitions use types to identify legal sets of data to be sent or retrieved. XML Schema (XSD) is used to define composite types and elements describing the data structures to be used.

- *XPath.* Data manipulation within a BPEL program requires addressing individual parts of a message and copying data from literals, variables, or variable contents to other variables or variable contents. To address the individual elements of an XSD-typed variable, the XPath language is used.

It is assumed that the reader is familiar with XML, XML Schema, and XPath. The four other standards are shortly introduced in the following. For a thorough description of the individual parts of a Web Service-based application, the reader is referred to [WCL$^+$05].

2.1.2.1 SOAP

SOAP [soa07] is a protocol for exchanging messages between peers in a distributed environment. SOAP is based on XML and defines a standardised message format, a processing model, a set of conventions defining how to map application data into messages, and a mechanism for binding messages to network protocols for transport. These are described in the following.

Standardised Message Format

SOAP specifies how information is packaged into a standardised XML document (a SOAP Message) to be transported in a communication. A SOAP message consists of a SOAP envelope, which in turn contains an (optional) header, and a (required) body. The header contains meta-data with instructions on how to process the message (see processing model). The actual application data is placed in the SOAP body (see mapping conventions).

Processing Model

The SOAP processing model defines *roles* for senders and receivers of SOAP messages, specifying which parts of a message must be processed by a role. In particular, the blocks making up a SOAP header may contain attributes indicating the intended use of this header block, i.e. operations to be performed by each node the SOAP message passes through. The payload of each header block is an XML fragment in its own namespace, and is thus not defined by SOAP.

Mapping Conventions

The SOAP specification contains a set of conventions for mapping application data into the SOAP messages, for example for specifying a remote procedure call. A mapping specifies an *interaction style* (RPC and document) as well as *encodings* (literal and SOAP encoding). The interaction style defines how the message is structured. In RPC (Remote Procedure Call) style, the SOAP body contains a complete method invocation with the operation name and parameters. In document style, the SOAP body contains a business document. The encodings, on the other hand, define how to encode application data into XML. The literal style does not perform any encoding (i.e. the data is already in XML format), while the SOAP encoding style defines a specific encoding format (for example, how to convert arrays, structured data types, and simple types like double and string into XML).

Network Protocol Bindings

Through bindings, SOAP specifies how to transmit SOAP messages over the network. SOAP includes two bindings out of the box: HTTP and SMTP. Through extensions, other protocols can also be used. It is important to note that in general, the target of a message — i.e., the address — is not part of a SOAP message. Instead, addressing is performed by the transport protocol, and the target address — an URL in case of HTTP, or an email address in case of SMTP -- must be specified outside of the SOAP message. However, a specialised header — for example, specified using the WS-Addressing standard — may be introduced to assume this task.

2.1.2.2 WSDL

The Web Service Description Language (WSDL) [CCMW01] is an XML-based language used to describe the functional interface of Web Services — what they can do, where they are located, what kind of data they expect and send out, and in which format. WSDL thus combines features commonly found in Interface Definition Languages (IDLs) with access and location information.

A WSDL documents consist of two parts: An abstract and a concrete part. This structure is shown in figure 2.2.

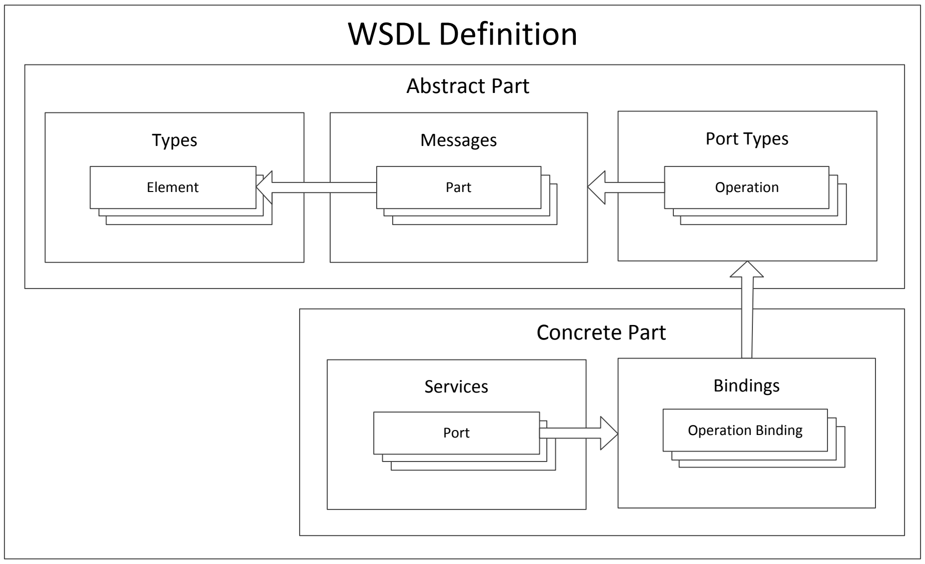


Figure 2.2: WSDL Definition Structure

All elements are required to define an executable and usable Web Service. The abstract part consists of three elements:

- The *Types* element contains basic data structure definitions, such as types of messages. The types are usually defined in XML Schema.
- Each of the *Message* elements is used to define a message to be sent or received; each message consists of *parts* which are usually typed with XML Schema types, either standard or newly defined in the *Types* section.

- Finally, each of the defined *PortType*s represents an interface with a list of operations which may be invoked on a service, each specifying input, output, and possible fault thrown by the operation. The input, output, and fault elements are specified as messages. Depending on which elements are present, an operation is inbound, outbound, or both. Usually, either *one-way* operations or *request-response* operations are used: The first receives a message from a client without returning a result; the second receives a message from the client and sends back an answer.

With types, messages, and port types specified, a client knows which operations are available, which messages with which parts must be sent in order to invoke an operation, and which messages with which parts are to be expected back (if any). However, the description is still abstract in the sense that the actual location of the service and the wire format of the messages are yet unspecified. This information is contained in the concrete part of a WSDL service description, which consists of two elements:

- A *Binding* element specifies how to bind a port type to a data transmit format; for example, use of SOAP over HTTP to transmit the actual data.

- The *Port* elements connect a binding with a port type, and additionally add the location of the port, mostly in terms of a URL. A port, in turn, belongs to a *Service* element which, finally, defines a service consisting of ports and associates it with a name.

Through this structure, the *Service* element contains the information from all parts and serves as the top-level element for the traversal of the definitions in a WSDL file.

2.1.2.3 WS-Addressing

As already noted in the previous sections, routing messages to the correct Web service and, in the case of asynchronous callbacks, back to the original sender (or a different Web Service, for the matter) requires additional addressing information to be included in the exchanged messages. The Web Service Addressing standard (WS-Addressing [wsa04]) provides such means by defining the notion of a service endpoint and how to encode addressing information in a message (specifically, a SOAP message).

The three key characteristics of WS-Addressing are protocol independence, asynchronous communication, and stateful, long-running transaction. Here, we are mainly concerned with the second point, as the first is out of scope of this thesis and the third is addressed by BPEL as well.

Asynchronous communication is concerned with the problem of being able to reply to a partner whose address is not known during design time. The key headers WS-Addressing introduces to SOAP headers to enable asynchronous callbacks are the following.

- `To` header. This header contains the address of the target endpoint. For example, if HTTP is used, this header contains an URL.
- `ReplyTo` header. The ReplyTo header contains a complete endpoint reference to send the reply message to. It must be present in the first message of a request-reply operation.
- `Action` header. This header contains an *identification URI*, defining the semantics of the message (in general, the operation invoked).
- `MessageID` header. This header contains an ID which uniquely identifies the message which it is part of. Specifying a message ID allows creating relationships between messages. This header is required in the first message of a request-reply operation.
- `RelatesTo` header. This header is the counterpart of the MessageID header and must contain the MessageID of the previous message in a message exchange. The `RelatesTo` header has an attribute specifying the communication relationship with a code (the `RelationShipType`), which is defined in the WS-Addressing specification. For example, a reply in a request-response operation carries the attribute `Reply`, which is also the default value if no attribute is specified.

Note that including WS-Addressing information alone is not sufficient for enabling asynchronous callbacks to this address; the target service invoked (and its middleware) must understand and support WS-Addressing to enable this feature.

2.1.2.4 BPEL

The ability to integrate, or compose, existing services into new services is, arguably, the most important functionality provided by SOAs [JMS03]. At its core, composition of services should allow creating, running, adapting, and maintaining services which rely on other services in an effective and efficient way [SD05].

The Business Process Execution Language (BPEL) has been created with the purpose of achieving this goal. BPEL is an XML-based programming language which allows the specification of both choreographies and orchestrations; only the latter are executable. Artefacts written in BPEL are closely integrated with other standards of the Web Service family, in particular WSDL, XML Schema, and XPath.

BPEL programs revolve around the following key concepts:

Partner Services

A BPEL process is intended to compose services. To achieve this goal, services must be identified and invoked; BPEL requires them to be described in WSDL.

Likewise, the BPEL process itself is invoked by another party to start the execution; through recursion, a BPEL process is a Web Service itself and must also be described in WSDL. A Web Service which interacts with the BPEL process in any way is called a *partner*.

BPEL does not differentiate between a client (which calls the BPEL process) and a Web Service which is invoked by the BPEL process, as an invoked Web Service may become a client itself when asynchronous callbacks are used. Instead, the relationship between a partner and the BPEL process is described in an abstract way by using the notion of *partner link types*. A partner link type contains at least one and at most two roles -- one role being played by the process and one being played by an external service.

Partner link types are instantiated to *partner links*, in which the roles are assigned: The process itself may take up a role by specifying it as `myRole`, or assign it to an external partner with `partnerRole`.

Workflow Specification

The BPEL business logic consists of a series of activities which are executed in sequence or in parallel. Activities are either basic or structured. Basic activities include the invocation of other Web Services or receiving incoming invocations; structured activities are used to define the relationship between activities. Finally, BPEL introduces the concept of handlers which may be attached to the process or a local scope to deal with exceptions, events, and compensation.

Many activities deal with variables or contents of variables. As BPEL is based on WSDL and XML Schema, types of variables are (usually) defined as WSDL messages or XML Schema types. XML Schema can thus be seen as the de facto type system of BPEL, and any variable data is literal XML which may be validated against an XML Schema. Variables may be declared on the process level, or on the level of scopes (see below).

Basic activities in BPEL can be separated into communication activities and other basic activities. The important ones in this context are the communication activities:

- The `receive` activity waits for a message from a partner; part of the receive specification is the partner link and the operation from the corresponding port type to expect.

- The `invoke` activity invokes a partner. Likewise, it is linked to a partner and an operation. The invoke activity can be used for both one-way and request-response operations, i.e. it is able to wait for a return value.

- The `reply` activity sends an answer for a message previously received with a receive activity.

Another important activity is the `assign` activity, which deals with data manipulation. An assign may contain a number of `copy` statements, which copy data from a right-hand side to a left-hand side. While the latter will always be a variable or message part, the right-hand side may include arbitrary statements; for example, calculations done with the help of XPath expressions.

Structured activities determine the order in which enclosed activities are executed. Structured activities may be nested in arbitrary ways, i.e. include basic or other structured activities. BPEL includes activities for sequential control (`sequence`, `switch`, and `while`), concurrent activity execution (`flow`), and reaction to events (`pick`).

The activities for sequential controls are well-known from conventional structured programming languages:

- The `sequence` activity represents a normal, sequential flow of execution: All enclosed activities will be executed one after another in the order they are specified. Most BPEL processes will use a sequence as the root activity.

- The `switch` activity introduces conditional behaviour. It contains a series of ordered case branches, each including a *branch condition*, and an optional *otherwise* branch, which is taken if none of the branch conditions hold true. The case branches are tested in the order in which they occur in the source code.

- The `while` activity repeats an enclosed activity until a condition no longer evaluates to true.

Support for concurrent activity execution is provided by the `flow` activity. This activity is the most powerful construct in the BPEL language. All activities inside a flow activity are executed simultaneously once the flow activity is executed.

Lastly, the `pick` activity allows local reactions to a set of events. When the activity is executed, it waits for the occurrence of `one` of these events, and executes the activities specified for this event.

Closely related to structured activities is the `scope` construct. A scope defines the behavioural context for a certain part of the BPEL document; it does not by itself impose any execution semantics on its children. However, handlers may be attached to the scope, which is discussed in the next paragraph.

Exceptions, Events, and Compensation

BPEL defines the concept of *handlers* to be able to deal with exceptions, events, and compensation.

Exception handling, termed *fault handling* in BPEL, follows the idea of object-oriented programming. To deal with faults, a *fault handler* may be associated with a scope (or with the process itself). A fault handler catches faults which are raised in a scope. If a fault occurs, processing of the scope is stopped, and a matching catch block is executed, which may contain arbitrary activities. A fault may be thrown by various means: The first are communication problems, the second are faults due to an incorrect BPEL specification, and finally, the BPEL `throw` activity allows for explicitly throwing a fault.

BPEL processes allow the specification of *event handlers* in the process itself and every scope. An event handler, as the name implies, handles events, which occur asynchronously while the main process logic is running. As a consequence of such an event, arbitrary activities may be executed, just like in the normal flow of the process.

An event handler is active as long as the scope it belongs to is active. In case of a global event handler, i.e. one defined in the process itself, the event handler stays active during the complete lifetime of a business process instance. An event handler cannot start a process instance; the instance must already be running.

There are two types of events which may be specified:

- *Message Events.* A message event occurs when a certain incoming message is received by the process.

- *Alarm Events.* An alarm goes off after a certain duration, or at a specified time.

It is important to note that events may occur multiple times during the runtime of a BPEL instance, but also not at all.

Finally, *compensation* in a BPEL process allows undoing successfully completed work in a certain scope. A compensation handler attached to a scope becomes active as soon as the scope has completed successfully; it can later be invoked by means of a `compensate` or `compensateScope` call.

A compensation handler for a scope contains arbitrary activities which undo the regular work of the scope. Note that the invocation is only allowed from the fault handlers and compensation handlers of the immediately enclosing scope. Furthermore, the process as a whole may also have a compensation handler, which undoes the complete business process. This compensation handler must be invoked by platform-specific means.

Correlation

Once a BPEL process instance is running, it sends out messages to partners, and expects answers in return. The BPEL middleware must be able to route such messages back to the appropriate instance, which is especially difficult when dealing with asynchronous callbacks. In traditional middleware solutions, an ID token was generated and maintained (added to messages and extracted from messages) by the middleware. BPEL, on the other hand, deals with loosely coupled, distributed Web Services not under the control of a specific middleware. Therefore, BPEL introduces the (optional) correlation concept to achieve the same thing by using the application data itself.

Correlation allows BPEL designers to decide which parts of the messages sent out to and received from partners constitute the ID of the message exchange, and thus, how to map message exchanges to a concrete business process instance. For example, when dealing with a Web Service for ordering items, the Web Service might give out an order number as part of the first exchange, which can then be used to correlate subsequent messages with this order number to the same business process instance.

As a BPEL process may deal with many different partners, multiple IDs may need to be created, which nevertheless point to the same process instance. In BPEL, correlation is based on two concepts: *properties* and *correlation sets*.

- A *property* has a name and type, and is mapped via *property aliases* to parts of the messages which might be sent or received as part of the interactions of a BPEL process. A property represents one part of an ID for a message exchange.
- A *correlation set* consists of multiple properties, and represents the complete ID of an application-level conversation with a process instance. A BPEL process may have multiple correlation sets for different sets of messages sent and received.

Correlation sets are used to associate messages with a business process instance. An initial message *initialises* the correlation set, i.e. defines the values of each property. The partner that sends this message is called the *initiator* of the set. All following messages must carry the identical values in the properties of the set to be routed to the given instance. All partners that use only follow-up messages are called *followers* of the set.

To use a correlation set, it must be attached to invoke, receive, and reply activities. The first message sent or received by one of these activities initiates the set; this fact must be specified as part of the activity.

2.2 The Unified Modeling Language

The Unified Modeling Language (UML) is a (visual) language for describing software designs, and is considered to be the de facto standard for modelling software systems. Its origins lie in three distinct methods which have been combined in 1994: The Booch Method [Boo93], the Object Modeling Technique [RBL$^+$90], and Objectory [Jac92]. UML has since been standardised under the umbrella of the Object Management Group (OMG); the latest version available is version 2.3 [OMG10b].

The aim of the UML is providing a way for developers to model software systems. For this purpose, the UML offers the ability to graphically describe certain aspects of a system; for example, one aspect might be the static structure, another might be the behaviour of a certain component. The UML provides several *diagram types* for each of these purposes, which each show a certain view on the modelled system. Taken together, a complete model of the system emerges.

The following three sections highlight the aspects of the UML which are important in this thesis. The first discusses the three most important model elements referred to in the remainder of the thesis. The second introduces UML *profiles*. Finally, section 2.2.3 briefly covers model serialisation.

2.2.1 Selected UML Modelling Elements

As indicated above, UML models may not only contain elements from various abstraction levels of the system, but also for different phases of the development process (for example, use case and deployment artefacts). In fact, the UML meta-model consists of over 250 classifiers to be used in 14 types of diagrams. The contributions in this thesis affect three diagram types and (part of) their model elements, namely composite structures, activities, and protocol state machines. In the following, a short introduction into the relevant elements is given.

2.2.1.1 Composite Structures

A first important concept for the design of complex systems are *composite structures*, i.e. the specification of elements which contain inner structure only visible on a relevant level of abstraction. An example for a class with composite structure is shown in figure 2.3.

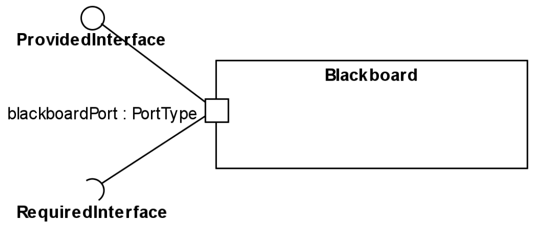


Figure 2.3: Structured Class Example

The main elements of a composite structure are the following:

- *Structured Classes.* A structured class (or more specifically, a class as extended by the `StructuredClasses` package) is used to denote classes which have internal structure and ports. The graphical representation is the same as a standard class used in a class diagram, i.e. a solid line rectangle with the class name written in the centre.
- *Ports.* A port offers an interaction point between an element and its environment. In the context discussed here, a port is used to denote the fact that the class offers functionality to the outside. The port thus effectively hides the inner structure and implementation of this functionality. A port is drawn as a small square on the boundary of the class; the name and multiplicity are written near the square.
- *Required and Provided Interfaces.* A port may have provided and required interfaces. The provided interfaces describe which requests the structured class is able to handle through the given port, while the required interfaces

specify the requests the class itself may issue through the port to connected elements. A provided interface is denoted through the lollipop notation, while a required interface is shown as a semi-circle.

- *Port Types.* Declaring a port implicitly defines a type of the port which implements the provided interfaces; an instance of this type is instantiated to represent a port at runtime. However, the port type may also be explicitly declared, in which case it must implement at least the provided interfaces of the port. A port type is denoted as text next to a port.

The SoaML profile defined in the next section uses composite structures as the basis for the specification of SOA-based systems and adds a few requirements of its own.

Note that the definition of composite structures in the UML includes the concept and representations of *collaborations*, which are not used in this thesis and are therefore left out here.

2.2.1.2 Activities

The previous section has shown how to denote a structured class and its ports; the latter representing entry points into the behaviour of the class. Behaviour of classifiers can be specified in UML using various methods such as interactions, state machines, or activities. In this thesis, the latter is used to model service behaviour.

Activities can be used for several purposes in UML. The simplest is a procedural description of work performed as a result of a method call. Another is the specification of a workflow, for example of a business process; in this case, events external to the system trigger the execution. In the first case, the activity starts directly with an execution of actions, while in the second, the activity contains actions which react to triggers.

An example for an activity is shown in figure 2.4. Activity modelling is based on the following main concepts:

- *Action Nodes.* The fundamental building blocks of activities are action nodes, or actions for short. An action describes a certain unit of executable functionality, which may also use input data and produce output data. The UML defines various kinds of actions. Communication actions include `AcceptCallAction` (for receiving a call request), `CallOperationAction` (for sending an operation call), and `ReplyAction` (for replying to an operation call); other important actions are `RaiseExceptionAction` (for throwing an exception) and `OpaqueAction` (which has an implementation-specific semantics). Fundamentally, an action is denoted as a rounded rectangle, but the actual representation depends on the type of the action.

- *Control Flow.* The actions which are specified as part of an activity are connected via control flows. Using control flows, the sequential behaviour

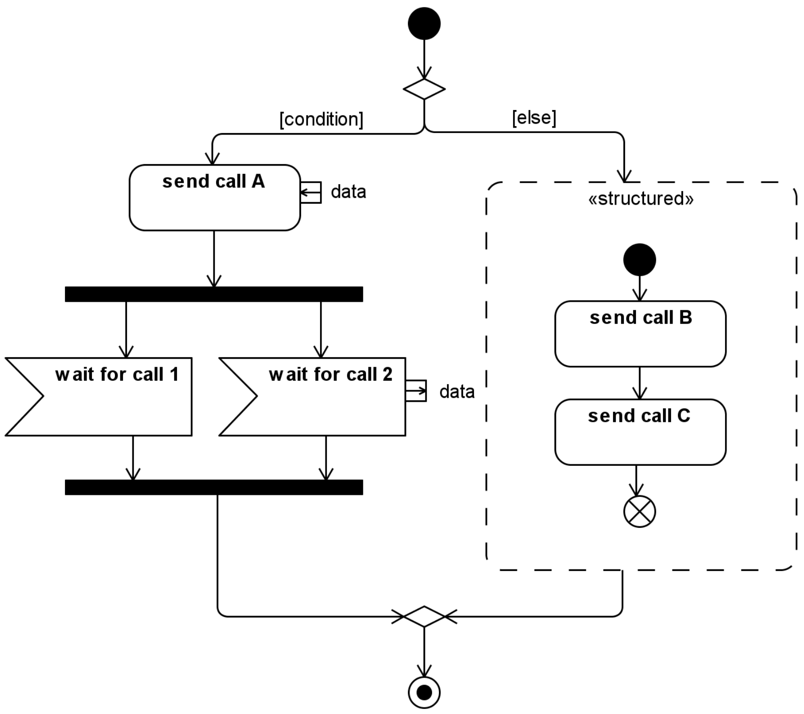


Figure 2.4: Activity Example

of an activity is specified. A control flow is an edge with an arrow, and indicates that the action on the arrow side of the edge is executed after the action on the non-arrow side of the edge. A sequence of actions combined using control flows is called a *flow* in short.

- *Control Nodes.* To coordinate the flow in an activity, control nodes may be used. There are five control nodes; three are used for denoting the start and end of an activity or part of an activity, while the other two are used to split and combine control flows (i.e., creating non-sequential behaviour). The simplest control node is the `InitialNode`, which indicates the start of an activity. The end of an activity is denoted by the `ActivityFinalNode`; a `FlowFinalNode` is used to end only the current flow, not the complete activity. For splitting control flow, `DecisionNode` and `MergeNode` are used; a decision node splits the flow into several ones of which only one may be executed, while the merge node combines them again. Another possibility is allowing several flows to run in parallel. The `ForkNode` is used to start multiple such flows, while a `JoinNode` is used to wait for all flows to return.

- *Nesting.* An activity may include nested elements to further structure the behaviour. First, an activity may invoke other activities by means of a `CallBehaviourAction` call. Secondly, an activity may contain `StructuredActivityNode` nodes, which are owned by the parent activity for the sole purpose of nesting. A related concept is the `InterruptibleActivityRegion`, which defines a certain area in the activity which can be interrupted by control flows leaving the region.

- *Object Flow, Object Nodes, and Pins.* Activities may not only be used to describe control flow, but data (object) flow as well. `ObjectFlow` edges are used to model the flow of data from or to `ObjectNode` elements. An object node may contain a value corresponding to its type; the flow indicates transfer of a value to another object. A specialised short-hand notation for object nodes is the `Pin`, which is directly attached to an action and models input (`InputPin`) required by the action or output (`OutputPin`) provided by the. Pins may again be linked with object flows, although, as detailed in the next chapter, an indirect variable-based data flow is possible as well.

- *Exception Handling.* The last relevant concept from activity modelling is exception handling, which is twofold. First, UML provides the `ExceptionHandler` element, which is used to denote an element which contains behaviour to execute if an exception occurs in a protected node. The protected node is linked to the handler may means of a special edge with a lightning-bolt notation. An exception may be thrown by means of a `RaiseExceptionAction`, which leads to the execution of the attached exception handler.

The UML specification defines additional concepts such as looping, expansion regions, or streaming actions. As these are not relevant for the extensions provided in this thesis, they are not further discussed here.

2.2.1.3 Protocol State Machines

The last model element used in this thesis are Protocol State Machines (PrSMs). PrSMs are restrictions of state machines which are used to express usage protocols; for example, compound transitions, sub-state machines, and composite states are not allowed.

PrSMs are always defined in the context of a classifier such as a structured class or a port: The aim of PrSM is the specification of call sequences, i.e. which operations of the classifier can be called in which state and under which condition. As such, PrSMs formalise the interface of classes.

An example of a protocol state machine is shown in figure 2.5. The following elements are key to protocol state machines:

- *States.* A state of a protocol state machine models a defined situation a behaviour is in. The state may be static, i.e. the behaviour is waiting for an

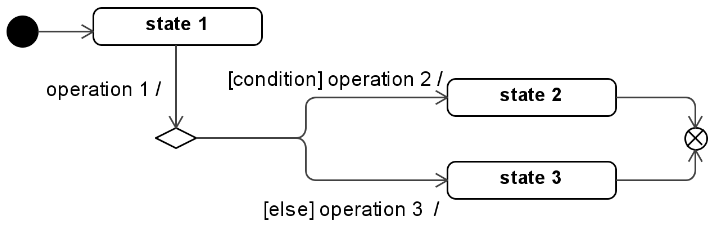


Figure 2.5: Protocol State Machine Example

external event, or dynamic, i.e. the behaviour is performing a calculation or is preparing a communication. A state is denoted as a rounded rectangle with the state name inside.

- *Protocol Transitions.* Each transition between states in a protocol state machine is a `ProtocolTransition`. A protocol transition may contain pre- and post conditions; it may also be associated to an operation of the context classifier through a `Trigger` with a certain event, for example, a `ReceiveOperationEvent`. A protocol transition may not contain an effect action. A transition is denoted as a solid open arrow between two states.
- *Pseudo States.* State machines may contain different kinds of pseudo states. As in activities, there is pseudo node for indicating the start of the protocol (`Initial`) and for denoting the end (`Exit`). Furthermore, the `Terminate` pseudo state aborts the execution of the complete PrSM. For modelling branches, the `Choice` pseudo node is used.

The description of protocol state machines concludes the introduction of the relevant UML modelling artefacts for this thesis. For more information on UML, the reader is referred to [OMG10b].

2.2.2 MOF, UML, and Profiles

One of the aims of this thesis is creating the ability to model service-oriented systems in UML, and thus, the introduction of a domain-specific, graphical language for SOA modelling. There are basically two options for creating such a language: The first is adapting and extending the UML itself to SOA concepts; the second is using the *profiles* extension mechanism provided by the UML. In the following, these two alternatives are elaborated.

Adapting and extending the UML requires changing the definition of UML. The UML specification is modelled using the Meta Object Facility (MOF), a platform-independent meta-modelling framework defined by the OMG.

MOF is defined as a layered architecture. Each layer (also referred to as a *meta-level*) contains instances of the elements defined in the level above it, and may define the meta-elements for the elements in the level below it. A special role is given to the uppermost level, which contains elements defined in itself for bootstrapping.

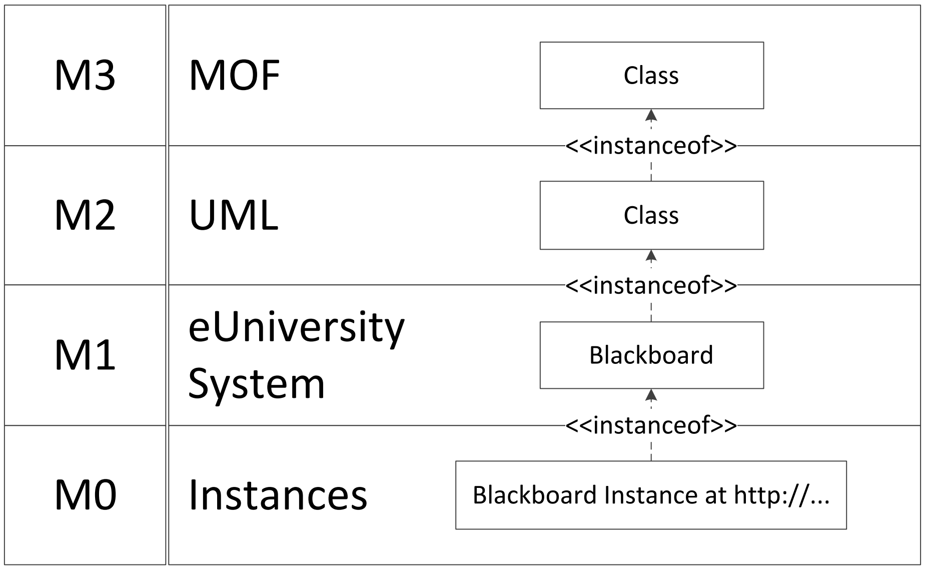


Figure 2.6: MOF Layers Example

An example for the layered approach of MOF is shown in figure 2.6, which shows the definition of the Unified Modeling Language in MOF, concrete systems modelled in UML, and instances of these systems. The figure thus shows four levels, named M3 to M0. The first level (M3) contains MOF, which is defined in itself. The second level (M2) contains UML, which is defined in terms of MOF. Concrete models of software systems, like for example the eUniversity system in the case study, are defined in terms of UML and thus reside on level M1. Finally, the classes of the eUniversity system are instantiated at runtime, which takes place in level M0.

The first option thus consists of changing the MOF meta-model of the UML according to the requirements of SOAs. This is discussed in the next section.

The second option does not require changes to the meta-model of the UML. The UML meta-model (residing in M2 in the above figure) already contains an extension mechanism which is separate from the MOF layers. This mechanism allows for the creation of *profiles*, which form a conservative extension to the UML and are only defined in the terms allowed by the UML specification. This is discussed in section 2.2.2.2.

Finally, both options may be combined, which will be shown in section 2.2.2.3.

2.2.2.1 MOF Meta-Models

As mentioned above, the first option of extending the UML is by extending, and possibly adapting, the UML meta-model itself. As the UML meta-model is modelled in MOF, any changes need to be done on the MOF level.

This is referred to as a *heavyweight* extension of the UML. Heavyweight extensions have the following benefits:

- Using MOF allows using the full power of meta-modelling to precisely and succinctly define relationships between modelling elements, which includes constraining extended classes.
- The definition and naming scheme of the newly introduced meta-model can follow the UML model more closely, making it more readable for developers with experience in UML.
- A complete meta-model may also serve as a domain definition, fully specifying the concepts and relations in a standard syntax. This also increases understandability for software engineers.

On the other hand, heavyweight extensions are not risk-free. Firstly, changes on this level are *completely unrestricted* and may even completely replace core UML concepts, altering the semantics to a point where knowledge of the UML is no longer useful for the newly introduced concepts.

A second problem associated with a heavyweight extension is tool support. As a MOF extension effectively creates or changes model elements directly, each tool needs to be specifically adapted to support them, i.e. there is no generic out-of-the-box support for changes to the UML meta-model.

2.2.2.2 Profiles

The *profile* extension mechanisms of the UML is defined in chapter 18 of the UML superstructure [OMG10b]. Profiles define a mechanism for *adapting* a reference meta-model (which, in this case, is the UML specification) for a specific target domain. In particular, the elements defined in a profile may not contradict the semantics of the reference meta-model, and may not add new meta-classes in the UML meta-class hierarchy nor change existing meta-class definitions. In a sense, a profile has a *read-only* relationship with its reference meta-model, i.e. the meta-model may only be extended without changes.

A profile is thus not a first-class extension mechanism but rather allows a lightweight extension of the UML. Profiles are defined by using the following concepts:

- A `Profile` itself is a UML `Package`, which references the UML meta-model package and any additional required profiles.
- Special meaning is given to UML meta-classes by means of `Extension` with a `Stereotype`. A stereotype defines a new semantics for the extended

class, which is applicable if the stereotype is used on a corresponding class in a concrete model.

- A stereotype may have attribute definitions, which are referred to as `Tag Definitions`. The types used for tag definitions may be standard meta-classes or stereotypes. When a stereotype is applied in a concrete model, the corresponding attribute values are referred to as `Tagged Values`.

Lightweight extension of the UML have the following benefits:

- A profile is easy to define, as it only allows extensions of existing meta-classes and thus, inherits most of the semantics from UML.
- In general, the risks of changing the UML semantics to a point where the original intentions are lost are lower than in heavyweight extensions.
- Most UML modellers include support for profiles, which means instant tool availability for new concepts introduced through profiles.

The disadvantage of using profiles is the loss of the ability to change existing concepts.

2.2.2.3 Hybrid DSL Specifications

Interestingly, many existing UML profile specifications do not introduce their profiles directly as a set of stereotypes and tag definitions. Rather, they define a MOF meta-model for the extension, later mapping the newly introduced meta-classes and meta-attributes to stereotypes and tag definitions.

Using this approach, the constraints of the profile mechanism and the extended language (i.e., UML) must already be considered during the definition of the meta-model extension. However, if successful, the resulting specifications combine the best of two worlds:

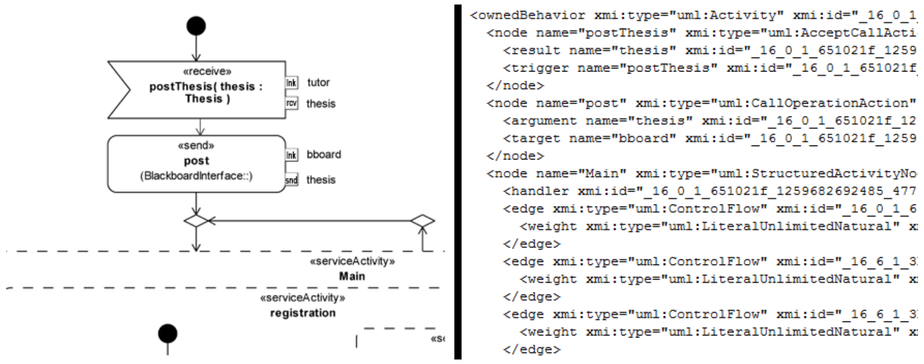
- Using MOF to define the meta-model for the new language allows for a precise and technically unrestricted definition of the extension; the description can follow the UML standard closely to increase understandability.
- Using the profile mechanism to define a corresponding profile has the advantage of tool support, and thus immediate applicability.
- In rare cases, the profile and the meta-model might disagree, indicating a missing or problematic feature of the UML meta-model. In this case, the disagreement is already a precise specification of the problem.

Examples of profiles with this approach include SoaML ([OMG09b]) and MARTE ([OMG08a]), both of which are on their way to becoming OMG standards. In fact, the first profile makes use of a disagreement between profile and meta-model to point to a new extension required in the UML for modelling service-oriented systems.

2.2.3 Model Serialisation

The principle of graphical representation lies at the core of UML. However, to be able to store and exchange UML models, a machine-readable serialisation format is required. UML 2 models are serialised in XMI according to the rules specified by the MOF 2.0 XMI Mapping Specification (see [OMG07]). In practise, however, different tools implement different versions of this specification, or define their own format altogether.

With the introduction of the Eclipse Model Development Tools project (MDT) [Ecl10c], this field has seen some consolidation. The MDT project defines the complete UML2 meta-model based on the Eclipse Modelling Framework (EMF); many tools are now based on this model and thus use the same serialisation format. The UML2-EMF-XMI format of Eclipse can be written and read by the Eclipse UML2 tools, read and written by the Rational Software Architect [IBM09] modelling tool by IBM, and exported by many other tools, for example by MagicDraw [NoM10].



```
<ownedBehavior xmi:type="uml:Activity" xmi:id="_16_0_1
  <node name="postThesis" xmi:type="uml:AcceptCallActi
    <result name="thesis" xmi:id="_16_0_1_651021f_1259
    <trigger name="postThesis" xmi:id="_16_0_1_651021f
  </node>
  <node name="post" xmi:type="uml:CallOperationAction"
    <argument name="thesis" xmi:id="_16_0_1_651021f_12
    <target name="bboard" xmi:id="_16_0_1_651021f_1259
  </node>
  <node name="Main" xmi:type="uml:StructuredActivityNo
    <handler xmi:id="_16_0_1_651021f_1259682692485_477
    <edge xmi:type="uml:ControlFlow" xmi:id="_16_0_1_6
      <weight xmi:type="uml:LiteralUnlimitedNatural" x
    </edge>
    <edge xmi:type="uml:ControlFlow" xmi:id="_16_6_1_3
      <weight xmi:type="uml:LiteralUnlimitedNatural" x
    </edge>
    <edge xmi:type="uml:ControlFlow" xmi:id="_16_6_1_3
      <weight xmi:type="uml:LiteralUnlimitedNatural" x
    </edge>
```

Figure 2.7: A UML Model and its Serialisation

An example of a serialisation of an UML2 activity diagram to EMF2-XMI is shown in figure 2.7. All tools presented in this thesis are based on this format.

It is important to note that the UML2-EMF-XMI format does *not* include any layouting information, such that manual layouts (which are quite common) are lost during serialisation and deserialisation.

2.3 SoaML

One of the contributions of this thesis is adding the ability to model service-oriented systems with UML by extending and adapting UML elements to the SOA view. Although the focus of the extensions in this thesis is on the behavioural part, behaviour can only be defined in the context of a system structure. Thus, a way of modelling the static part of a SOA system in UML is required as well.

A first step into this direction has already been taken with the UML profile *SoaML* [OMG09b], which is a draft OMG standard for describing the *static aspects* of service-oriented systems in UML. Although SoaML is currently in beta status, the concepts are considered valid and stable enough to be included in this thesis. Nevertheless, the final version of SoaML might differ from the concepts presented here.

Structural service modelling in SoaML employs the basic UML mechanisms for modelling composite structures as introduced in the last section. SoaML adds several stereotypes to the elements present in composite structures, and defines several constraints on these elements. In the following, the SoaML concepts relevant for this thesis are presented.

2.3.1 Participants

In SoaML, the basic unit for implementing service functionality is a *participant*. A participant can represent people, organisations, or system components, and may offer or require any number of services. SoaML introduces the stereotype ≪*Participant*≫, which is used to tag UML components to indicate that the component takes part in a SOA. A participant may be both a consumer and a provider of services, depending on the ports associated with the participant.

A SoaML participant may include UML ports, which are interaction points where services are provided or requested. Each of the ports of a participant is tagged with either ≪*Service*≫ or ≪*Request*≫. In the former case, the participant offers a service through this port, while in the second case, a service is requested on this port.

The behaviour of the provided services of a participant may be implemented by different means. The SoaML specification lists three possible ways of providing such behaviour:

- A first option is to implement each *provided operation* separately, i.e. by using an owned behaviour of the corresponding method. The behaviour may be realised by an interaction, activity, state machine, protocol state machine, or opaque behaviour.

- A second action is through the means of *event handling*. Using this concept, a participant is active in the sense that it already contains running behaviour and is thus able to react to incoming events using actions such as `AcceptCallAction`.

- The third options is delegation: A participant may also delegate incoming requests to inner parts in the composite structure. This allows for wrapping existing implementations, or for further decomposition.

The UML4SOA profile introduced in chapter 3 takes the second approach by providing workflows which are able to react to incoming events from the ports of the owning participant.

SoaML defines another concept, the ≪*MessageType*≫, which is relevant to this thesis. A message type is an extension of the UML meta-classes `DataType` and `Class`, and is used to represent information exchanged between participant requests and services. Message types may not contain operations or behaviours: The intention is to only allow data to be transported between services. Data can be represented through attributes and associations to other message types.

SoaML requires that all input and output parameters of operations defined in the provided and required interfaces of service and request ports must be typed with either simple types or message types.

2.3.2 Services and Requests

Ports of SoaML participants tagged with ≪*Service*≫ or ≪*Request*≫ may be typed with a simple UML interface (in which case, no protocol is assigned to the port), or with a ≪*ServiceInterface*≫, which is another concept defined by SoaML. A ≪*ServiceInterface*≫, which is an extension of an UML class, specifies the responsibilities of both the participant to which the corresponding port is attached, and the communication partner which is connected to the port.

For service interfaces used as types of ≪*Service*≫ ports, the semantics is as expected from the UML: Interfaces *implemented* by the service interface are *provided* by the participant; thus, the behaviour of the participant must implement them and provide the corresponding functionality. Interfaces *used* by the service interface are *required* by the participant, thus, the communication partner must implement them and provide the corresponding functionality.

When using ≪*Request*≫ ports, this situation is inverse: Implemented operations of the service interface are requested by the participant, and must be implemented by the partner. To avoid having to create two service interfaces due to this problem, SoaML introduces the notion of *conjugation*. In a ≪*Request*≫ port, the provided and required interfaces of a port type are *inverted*, which means that the port uses the port type instead of implementing it.

Note that when using service interfaces, provided and required interfaces are not specified on the port but on the port type, and are derived in the port. Furthermore, the behavioural protocol observed at the port is also attached to the service interface. This will become important in chapter 3 with the definition of service protocols.

2.3.3 SoaML Meta-Model

The meta-model for the four new SoaML concepts introduced above — participants, service ports, request ports, and service interfaces, is shown in figure 2.8.

The figure shows how the ≪*Service*≫ and ≪*Request*≫ stereotypes are derived from the `Port` meta-class, and redefines the type of `Port` with the ≪*ServiceInterface*≫ stereotype. The latter is an extension of the UML meta-

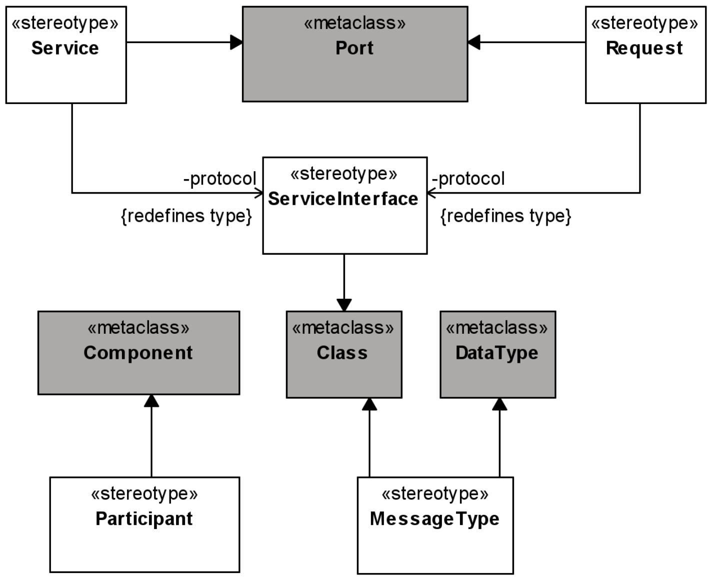


Figure 2.8: SoaML Meta-Model (Excerpt)

class `Class`. Furthermore, on the bottom, the stereotype ≪*Participant*≫ is shown as an extension of `Component`, and the ≪*DataType*≫ stereotype as an extension of both `Class` and `DataType`.

2.3.4 Case Study

SoaML has been used to model the static part of the eUniversity case study introduced in chapter 1. The architecture makes use of all of the SoaML stereotypes introduced above and is shown in figure 2.9. Note that the port types are associated with the ports with a usage link stereotyped with ≪*type*≫; this is a convenience notation used to clarify the relationship between a port and its service interface and is not a UML or SoaML concept. The figure shows only one participant — `ThesisManagement` — which has two ≪*Service*≫ ports and three ≪*Request*≫ ports. The service interfaces of the ≪*Service*≫ ports both implement and use operations, while the service interfaces of the ≪*Request*≫ ports are simpler and only implement operations.

The types referenced in the composite structure are all message types, and are shown in figure 2.10.

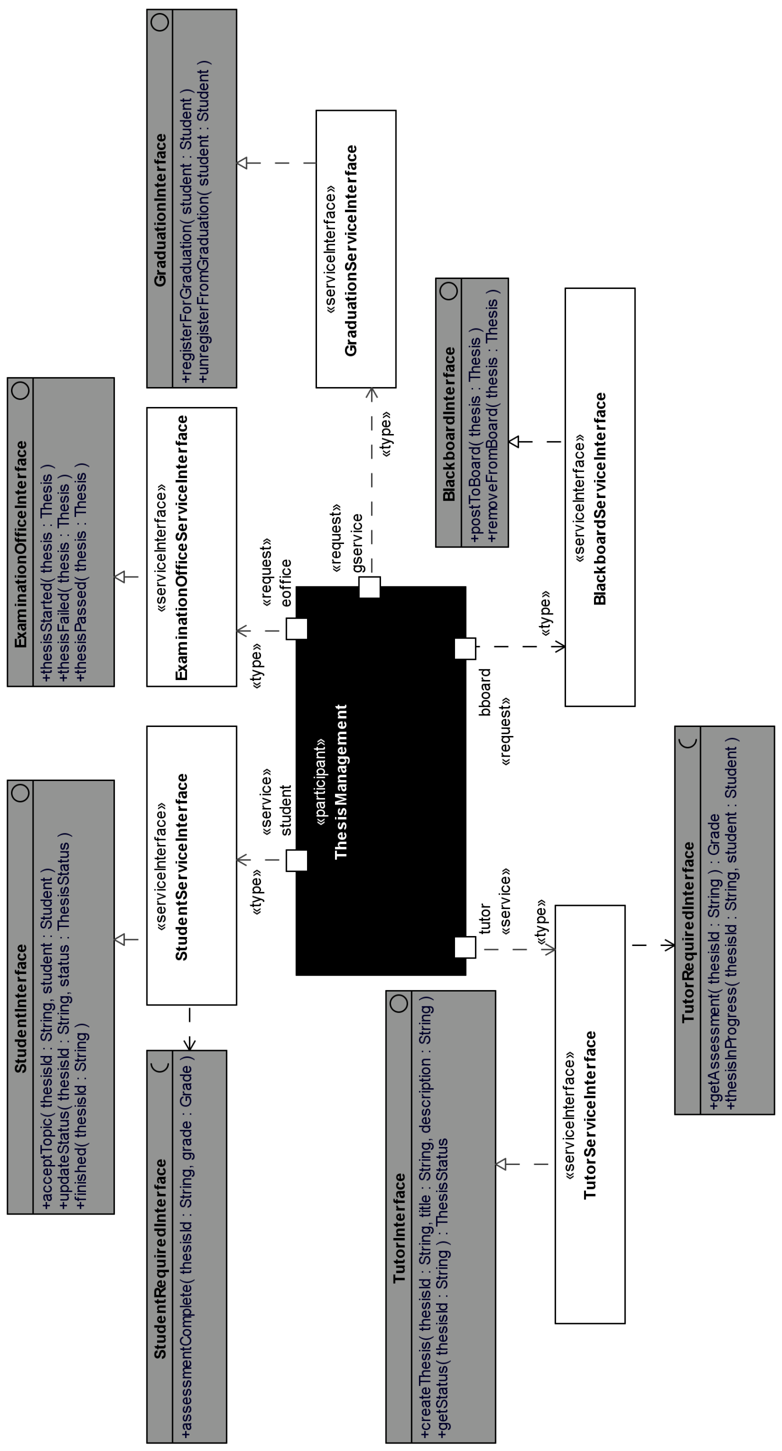

GraduationInterface
+registerForGraduation(student : Student)
+unregisterFromGraduation(student : Student)
«serviceInterface»
GraduationServiceInterface
ExaminationOfficeInterface
+thesisStarted(thesis : Thesis)
+thesisFailed(thesis : Thesis)
+thesisPassed(thesis : Thesis)
«serviceInterface»
ExaminationOfficeServiceInterface
BlackboardInterface
+postToBoard(thesis : Thesis)
+removeFromBoard(thesis : Thesis)
«serviceInterface»
BlackboardServiceInterface
«type»
«request»
gservice
«request»
eoffice
«type»
«type»
bboard
«request»
«participant»
ThesisManagement
«service»
student
«type»
tutor
«service»
«type»
StudentInterface
+acceptTopic(thesisId : String, student : Student)
+updateStatus(thesisId : String, status : ThesisStatus)
+finished(thesisId : String)
«serviceInterface»
StudentServiceInterface
StudentRequiredInterface
+assessmentComplete(thesisId : String, grade : Grade)
TutorInterface
+createThesis(thesisId : String, title : String, description : String)
+getStatus(thesisId : String) : ThesisStatus
«serviceInterface»
TutorServiceInterface
TutorRequiredInterface
+getAssessment(thesisId : String) : Grade
+thesisInProgress(thesisId : String, student : Student)

Figure 2.9: eUniversity Case Study: Static Model

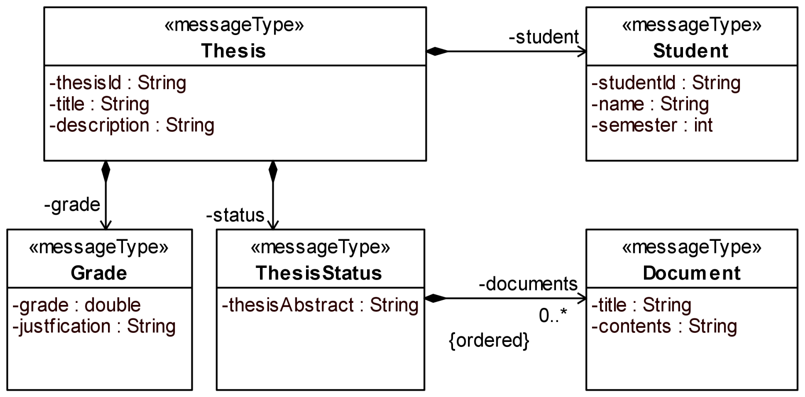


Figure 2.10: eUniversity Case Study: Data Types

2.4 Model-Driven Development

Model-Driven Development (MDD) [MCF03, Sel03], also called Model-Driven Software Engineering (MDSE), refers to a software development process which focuses on building *models* of software systems before actually implementing them. This principle is well-known in traditional engineering principles; in fact, in many cases the construction of complex structures such as buildings or bridges is not feasible without a model. MDD attempts to transfer the advantages of modelling to software engineering, in which case the benefits from a process based on models promise to be every greater: Through automation, models can be used for more than just communication purposes; in fact, they can be used to drive the system implementation or even replace the need to implement a system by hand.

The concept of model-driven development is independent of any concrete technology. Several frameworks for MDD have been proposed, the most prominent of which is the Model-Driven Architecture (MDA) by the OMG [Ric00]. MDA is closely related to other OMG standards such as the UML and MOF introduced in previous sections.

This thesis takes a wider approach to MDD without committing to any particular OMG or other standard.

The following subsections describe the idea of models in software development, the added benefits of being able to analyse models before implementing a software system, and finally the automatic generation of software artefacts and, in the end, executable code.

2.4.1 Models

It is difficult to arrive at a global, agreed-upon definition of a *model*, as models differ between application areas, abstraction level, and purpose. For example, one might consider code written in Java or C# to be a model of a software system, since it abstracts from the concrete machine instructions, instead allowing developers to think and write code in terms of classes, objects, and (virtual) method invocations. Another view might consider code written in such a language to be too concrete, and consider a graphical representation of a system, for example drawn using UML constructs, as a model later to be implemented in Java or another language.

In general, the definition of a model tends to concentrate as much on what is modelled as on what is not, i.e. the *implementation* which is abstracted away. Each of the two examples provided above is based on this view: The respective models are more general than their counterpart implementations, and thus further removed from any concrete implementation technique and more focused on abstract concepts. The term of *platform independence* is often associated with models, which reflects the attempt to model software independent of any one realisation option. An example of the inverse relationship of abstraction and platform dependence is shown in figure 2.11.

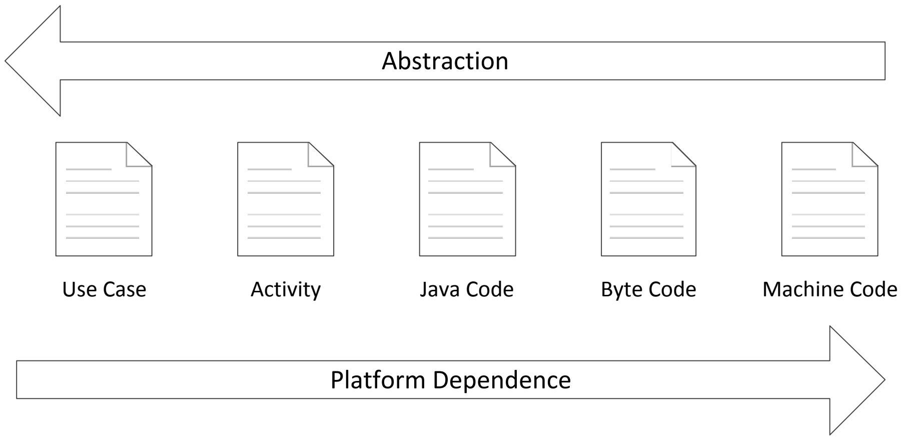


Figure 2.11: Model-Driven Development

Independent of the concrete model abstraction level, a model should be easy to specify, understand, and maintain, which means that the modelling technique must be tailored to the concrete problems at hand. This has given rise to the concept of domain-specific modelling languages (DSMLs). It is also important to note that using MDD is by no means a single-step approach: Models can be refined in a step-wise fashion, thus introducing many intermediate representations of the system with varying levels of details.

There is always a balance involved in models between being too generic —

and thus allowing too much freedom in implementation — and too specific, negating the benefits of using a model in the first place. The level of detail required depends on the target use of the model. For maximum usability, a model should at least be amenable to analysis, if not for code generation. These two ideas are discussed in the following.

2.4.2 Analysis on Models

Using models as an abstract representation of a computer system has the benefit of being more amenable to (formal) analysis, as fewer details need to be taken care of and the essence of the software behaviour can be extracted in an easier fashion. As models are available in an earlier phase of a software development process than the actual implementation, there is a higher chance of still being able to correct a system design if problems are found.

Analysis of software models has the aim of testing, or verifying, that certain properties hold in the modelled software. Such analysis can take the form of executing (parts of) a model to evaluate its runtime properties; another form is using rigorous mathematical techniques for verifying the qualitative or quantitative aspects of the system, i.e. aspects such as deadlock freeness or performance analysis and prediction in different usage scenarios.

In chapter 1, the Sensoria project has already been mentioned. Many results of this project lie in the area of model-driven verification by using formal methods; also, several of the tools developed in Sensoria address this area and have been integrated into the common tooling platform introduced in chapter 8.

2.4.3 Model Transformations and Code Generation

Model-driven development and its promise of automated development is closely linked to the domain of *model transformation*, which is in fact regarded as one of the core technologies for the realisation of model-driven development.

As noted above, model-driven development is not a single-step approach from a model to executable source code. Rather, multiple steps are possible where both the source and the target artefacts can be regarded as models of the software system. Thus, model transformation may refer to any translation from a source software artefact to a target artefact; the latter might even be a model of machine code.

Model transformations are defined in terms of domain *meta-models* instead of models. As shown in figure 2.12, a model transformation relates two meta-models to one another; execution of the transformation then results — depending on the direction — of a transformation of a concrete model compliant to one of the meta-models to a concrete model compliant to the other meta-model.

Model transformations, in general, are not concerned with generating code, but stay on the level of model instances. Thus, two additional steps are required on a technical level to acquire the source model from its native representation, and to emit the target model in the same way. These are referred to as deserialisation and serialisation, and are shown at the bottom of the figure.

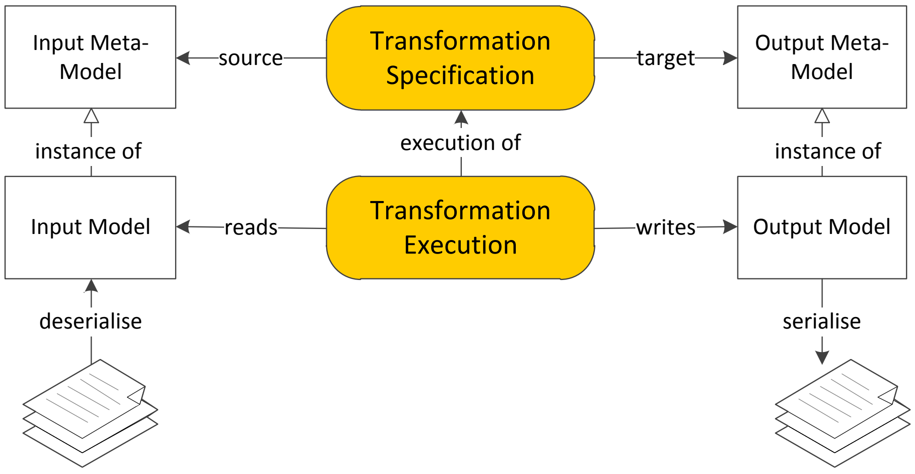


Figure 2.12: Model Transformations

It is important to note that MDD in itself does not define which models are considered read-only, and which are to be edited further. In figure 2.11, for example, a concrete development process might allow editing of all artefacts, or only of the artefacts higher than or on the same level as UML or Java. There are two fundamental views of MDD which affect this decision.

- The first view is that of MDD as a compilation step. In this view, a certain model is selected as the definitive implementation of the system. All other models on the way to machine code are generated and not to be read by humans. In this view, MDD is a natural extension, or new layer, on top of existing compilers.
- The second view is that of MDD as a helping hand in programming: Models do not contain the complete specification of a system, but are rather seen as a first step to generate code which is later extended in a lower-level programming language.

Note that in the first view, all information about the system resides in the model, as all changes are done in the model, and the code is re-generated after each change. In the second view, changes to the initially generated code lead to inconsistencies between the model and the implementation. Possible remedies include discarding the model once it has served its purpose, or attempting to keep both artefacts in sync. The latter, however, is a complicated topic and a research area of its own.

2.5 Modal I/O Automata and Interface Theories

Part of the specification of a high-level modelling language for SOA behaviour in this thesis is the definition of a formal semantics, which, in turn, offers the ability for formal verification of SOA systems modelled using this technique. As will be discussed further in chapter 5, the semantic domain chosen for the developed language are *modal input/output automata (MIOs)* [LNW07a]; furthermore, the verification methods used are based on *interface theories* [BMSH10].

In this section, MIOs as well as interface theories are introduced. A more thorough discussion of these topics may be found in [BMSH10].

2.5.1 Modal I/O Transition Systems

Among the most widely accepted methods for specifying behavioural properties of software are *input/output automata* [LT87, LT89], which have been introduced to specify the temporal ordering of events involving a component, explicitly taking communication aspects such as sending or receiving messages into consideration. Many variations of these automata have been introduced over the years; for example interface automata [dAH05], timed interface automata [dAHS02], or resource automata [CdAHS03].

At the same time, another aspect of interface behaviour has been studied: Modal automata [LT88b] explicitly address the difference between required and optional actions by using must and may transitions, which allow protocols and implementations to differ with regard to non-compulsory actions.

Recently, both the input/output and the may/must aspects of behavioural specifications have been integrated [LNW07a], giving rise to modal I/O automata (MIOs).

A modal transition system is characterised by the fact that it has *two* transition relations, indicating allowed (*may*) and required (*must*) behaviour. Here, we consider an extended version of the original modal transition systems [LT88b] by including a signature which distinguishes between *internal* and *external* actions.

Definition 1 (Modal Transition System) *A modal transition system (MTS) $S = (states_S, start_S, (ext_S, int_S), \dashrightarrow_S, \longrightarrow_S)$ consists of a set of states $states_S$, an initial state $start_S \in states_S$, disjoint sets ext_S and int_S of external and internal actions where $act_S = ext_S \cup int_S$ denotes the set of (all) actions, a may-transition relation $\dashrightarrow_S \subseteq states_S \times act_S \times states_S$, and a must-transition relation $\longrightarrow_S \subseteq states_S \times act_S \times states_S$. The pair (ext_S, int_S) is called the signature of S.*

An MTS S is called *syntactically consistent* if every required transition is also allowed, i.e. it holds that $\longrightarrow_S \subseteq \dashrightarrow_S$. From now on we only consider syntactically consistent MTSs. Moreover, we call an MTS S an *implementation* if the two transition relations coincide, i.e. $\longrightarrow_S = \dashrightarrow_S$.

Modal I/O transition systems [LNW07a] further differentiate between two kinds of external actions, namely *input* and *output* actions.

Definition 2 (Modal I/O Transition System) *A modal I/O transition system (MIO) S is an MTS with the set of external actions ext_S partitioned into two disjoint sets in_S, out_S of input and output actions, respectively. The triple (in_S, out_S, int_S) is called the signature of S.*

The notions of syntactic consistency and implementation also apply for MIOs.

2.5.2 Interface Theories

When modelling the behaviour of service- or component-based systems, two questions may be asked of a model:

- Does a certain behavioural model *refine* another model?
- Are two behavioural models *compatible* with one another?

Interface theories or interface languages are commonly used to precisely define these notions. Interface theories can be defined as follows (see [BMSH10]):

Interface theories are tuples $\mathcal{I} = (\mathcal{A}, \leq, \sim, \otimes)$ consisting of a specification domain $\mathcal{A}$, a reflexive and transitive refinement relation $\leq \subseteq \mathcal{A} \times \mathcal{A}$, a symmetric compatibility relation $\sim \subseteq \mathcal{A} \times \mathcal{A}$, and a partial composition operator $\otimes : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$. If two interfaces are compatible then their composition is defined, i.e. for all $S, T \in \mathcal{A}$, if $S \sim T$ then $S \otimes T$ is defined. Moreover, interface theories impose the following requirements on their refinement and compatibility relations:

- Preservation of compatibility under refinement:

$$\text{for all } S, T, T' \in \mathcal{A},\quad \text{if } S \sim T \text{ and } T' \leq T \text{ then } S \sim T'.$$

- Compositionality:

$$\text{for all } S, T, T' \in \mathcal{A},\quad \text{if } S \otimes T \text{ defined and } T' \leq T \text{ then } S \otimes T' \text{ defined and } S \otimes T' \leq S \otimes T.$$

These properties imply *independent implementability*, which is the basis for a top-down design of services and service protocols: In order to refine a given composed interface $S \otimes T$ towards an implementation, it suffices to independently refine S and T, say, to S' and T', respectively; then the refinements S' and T' are compatible and their composition (which is defined) refines the interface $S \otimes T$.

In this thesis, we shall always employ modal I/O transition systems as the domain for the interface theories discussed, varying the notions of refinement, compatibility, and composition.

2.5.2.1 Strong Refinement and Compatibility

In the following, we recall the standard definition of refinement for modal transition systems, cf. [LT88b]. The notion of refinement aims at capturing the relation between an abstract specification of an interface and a more detailed one, possibly an implementation of that interface. Thus, it allows for a stepwise refinement of an abstract specification towards an implementation.

The basic idea of *modal* refinement is that any required (*must*) transition in the abstract specification must also occur in the concrete specification. Conversely, any allowed (*may*) transition in the concrete specification must be allowed by the abstract specification. Moreover, in both cases the target states must conform to each other. Modal refinement has the following consequences: A concrete specification may leave out allowed transitions, but is required to keep all must transitions, and moreover, it is not allowed to perform more transitions than the abstract specification admits. The following definition of modal refinement is called *strong* since every transition that is taken into account must be simulated "immediately", i.e. without performing internal actions before.

Definition 3 (Strong Modal Refinement [LT88b]) *Let S and T be MTSs (MIOs, resp.) with the same signature. A relation $R \subseteq states_S \times states_T$ is called strong modal refinement for S and T iff for all $(s,t) \in R$ and for all $a \in act_S$ it holds that*

1. *if $t \xrightarrow{a}_T t'$ then there exists $s' \in states_S$ such that $s \xrightarrow{a}_S s'$ and $(s',t') \in R$,*
2. *if $s \overset{a}{\dashrightarrow}_S s'$ then there exists $t' \in states_T$ such that $t \overset{a}{\dashrightarrow}_T t'$ and $(s',t') \in R$.*

We say that S strongly modally refines T, written $S \leq_m T$, iff there exists a strong modal refinement for S and T containing $(start_S, start_T)$.

If both S and T are implementations, i.e. the must-transition relation coincides with the may-transition relation, then strong modal refinement coincides with (strong) bisimulation; if $\longrightarrow_T = \emptyset$ then it corresponds to simulation [Mil89].

Next, we introduce a composition operator on MIOs. When two protocols (implementations), each one describing a particular component, can communicate by synchronous message passing, we are interested in computing the resulting protocol (implementation) of the composed system.

Although composition can obviously be defined for MTSs, we directly give a definition for MIOs as this is our main interest.

It is convenient to restrict the composition operator to *composable* MIOs by requiring that overlapping of actions only happens on complementary types.

Definition 4 (Composability [LNW07a]) *Two MIOs S and T are called composable if $(in_S \cup int_S) \cap (in_T \cup int_T) = \emptyset$ and $(out_S \cup int_S) \cap (out_T \cup int_T) = \emptyset$.*

We now define composition of MIOs in a straightforward way by a binary partial function $\otimes$ synchronising on matching (shared) actions.

Definition 5 (Composition [LNW07a]) *Two composable MIOs S_1 and S_2 can be composed to a MIO $S_1 \otimes S_2$ defined by $states_{S_1 \otimes S_2} = states_{S_1} \times states_{S_2}$, the initial state is given by $start_{S_1 \otimes S_2} = (start_{S_1}, start_{S_2})$, $in_{S_1 \otimes S_2} = (in_{S_1} \setminus out_{S_2}) \cup (in_{S_2} \setminus out_{S_1})$, $out_{S_1 \otimes S_2} = (out_{S_1} \setminus in_{S_2}) \cup (out_{S_2} \setminus in_{S_1})$, $int_{S_1 \otimes S_2} = int_{S_1} \cup int_{S_2} \cup (in_{S_1} \cap out_{S_2}) \cup (in_{S_2} \cap out_{S_1})$. The transition relations $\dashrightarrow_{S_1 \otimes S_2}$ and $\longrightarrow_{S_1 \otimes S_2}$ are given by, for each $\rightsquigarrow \in \{\dashrightarrow, \longrightarrow\}$,*

- *for all $i, j \in \{1, 2\}, i \neq j$, for all $a \in (act_{S_1} \cap act_{S_2})$, if $s_i \overset{a}{\rightsquigarrow}_{S_i} s'_i$ and $s_j \overset{a}{\rightsquigarrow}_{S_j} s'_j$ then $(s_1, s_2) \overset{a}{\rightsquigarrow}_{S_1 \otimes S_2} (s'_1, s'_2)$,*
- *for all $a \in act_{S_1}$, if $s_1 \overset{a}{\rightsquigarrow}_{S_1} s'_1$ and $a \notin act_{S_2}$ then $(s_1, s_2) \overset{a}{\rightsquigarrow}_{S_1 \otimes S_2} (s'_1, s_2)$,*
- *for all $a \in act_{S_2}$, if $s_2 \overset{a}{\rightsquigarrow}_{S_2} s'_2$ and $a \notin act_{S_1}$ then $(s_1, s_2) \overset{a}{\rightsquigarrow}_{S_1 \otimes S_2} (s_1, s'_2)$.*

Composition of MIOs only synchronises transitions with matching shared actions and same type of transition, i.e. a must-transition labeled with a shared action only occurs in the composition if there exist corresponding matching must-transitions in the original MIOs.

A well-known problem occurs when composing arbitrary MIOs S and T: If for a reachable state (s, t) in $S \otimes T$, S in state s is able to send out a message a shared with T, and T in state t is not able to receive a then this is considered as a compatibility problem since S may get stuck in this situation. We want to rule out this erroneous behaviour by requiring that S and T must be *compatible.*

The following definition of strong compatibility is strongly influenced by [dAH05] and [LNW07a]. Intuitively, two MIOs S and T are compatible if for every reachable state in the product $S \otimes T$, if S is able to provide an output which is shared with T, i.e. is in the input alphabet of T, then T must "immediately" be able to receive this message (and vice versa).

Definition 6 (Strong Modal Compatibility) *Let S and T be composable MIOs. S and T are called strongly modally compatible, denoted by $S \sim_{sc} T$, iff for all reachable states (s, t) in $S \otimes T$,*

1. *for all $a \in (out_S \cap in_T)$, if $s \overset{a}{\dashrightarrow}_S s'$ then there exists $t' \in states_T$ such that $t \xrightarrow{a}_T t'$,*
2. *for all $a \in (out_T \cap in_S)$, if $t \overset{a}{\dashrightarrow}_T t'$ then there exists $s' \in states_S$ such that $s \xrightarrow{a}_S s'$.*

MIOs equipped with $\sim_{sc}$ and $\leq_m$ form a valid interface theory (see [BMSH10] for a proof).

2.5.2.2 Weak Refinement and Compatibility

The basic (strong) form of modal refinement requires that every transition that is taken into account must be simulated *immediately.* There are many application areas in which this definition is too strong (see [HL89]). However, it can be weakened by distinguishing between external and internal actions and allowing an external action to be enclosed in internal actions. In this case, we speak of *weak transitions.*

For denoting weak transitions, given a MIO S and an action $a \in ext_S$, we write $s \xrightarrow{a}{}^{*}_{S} s'$ iff there exist states $s_1, s_2 \in states_S$ such that

$$s(\xrightarrow{\tau}_S)^* s_1 \xrightarrow{a}_S s_2 (\xrightarrow{\tau}_S)^* s'$$

where $t(\xrightarrow{\tau}_T)^* t'$ stands for finitely many transitions, labelled with internal actions, leading from t to t'; possibly no action and in this case $t = t'$. Here and later on, the action τ always denotes an arbitrary internal action. Moreover, we write

$$s \xrightarrow{\hat{a}}{}^{*}_{S} s' \text{ iff either } s \xrightarrow{a}{}^{*}_{S} s' \text{ and } a \in ext_S, \text{ or } s(\xrightarrow{\tau}_S)^* s'.$$

Both notations are analogously used for may-transitions. Using this notion of weak transitions, we can define weak modal refinement.

Definition 7 (Weak Modal Refinement [HL89]) *Let S and T be MIOs such that $\alpha_{ext}(S) = \alpha_{ext}(T)$. S weakly modally refines T, denoted by $S \leq_m^* T$, iff there exists a relation $R \subseteq states_S \times states_T$ containing $(start_S, start_T)$ such that for all $(s,t) \in R$, for all $a \in act_S \cup act_T$,*

1. *if $t \xrightarrow{a}_T t'$ then there exists $s' \in states_S$ such that $s \xrightarrow{\hat{a}}{}^{*}_{S} s'$ and $(s',t') \in R$,*
2. *if $s \overset{a}{\dashrightarrow}_S s'$ then there exists $t' \in states_T$ such that $t \overset{\hat{a}}{\dashrightarrow}{}^{*}_{T} t'$ and $(s',t') \in R$.*

Weak modal compatibility *moderates* the usual notion of strong compatibility [LNW07a] by requiring that an output (issued by a may- or must-transition) must be accepted with a corresponding input (by a must-transition), which may, however, possibly be preceded and followed by internal actions.

Definition 8 (Weak Modal Compatibility [BMSH10]) *Let S and T be composable MIOs. S and T are called weakly modally compatible, written $S \sim_{wc} T$, iff there exists a relation $R \subseteq states_S \times states_T$ containing $(start_S, start_T)$ such that for all $(s,t) \in R$,*

1. *for all $a \in (out_S \cap in_T)$, if $s \overset{a}{\dashrightarrow}_S s'$ then $\exists t' \in states_T . t \xrightarrow{a}{}^{\triangleleft}_{T} t'$ and $(s',t') \in R$,*
2. *for all $a \in (out_T \cap in_S)$, if $t \overset{a}{\dashrightarrow}_T t'$ then $\exists s' \in states_S . s \xrightarrow{a}{}^{\triangleleft}_{S} s'$ and $(s',t') \in R$,*
3. *for all $a \in (int_S \cup ext_S \setminus shared(S,T))$, if $s \overset{a}{\dashrightarrow}_S s'$ then $(s',t) \in R$,*
4. *for all $a \in (int_T \cup ext_T \setminus shared(S,T))$, if $t \overset{a}{\dashrightarrow}_T t'$ then $(s,t') \in R$.*

Again, MIOs equipped with $\sim_{wc}$ and $\leq_m^*$ form a valid interface theory (see again [BMSH10] for a proof).

2.5.2.3 Hiding

The interface theories introduced in the previous sections are based on matching alphabets of the automata to be refined or verified for compatibility. The notion of *hiding* (see, e.g., [Mil89]), i.e. internalising a set of actions, has been introduced to be able to loosen this constraint in certain situations. We can define hiding as follows:

Definition 9 (Hiding) *Let* $S = (states_S, start_S, in_S, out_S, int_S, \dashrightarrow_S, \longrightarrow_S)$ *be a MIO, and* $X \subseteq ext_S$ *a set of actions. Then, hiding the actions* X *in* S *yields* $S \setminus X = (states_S, start_S, in_S \setminus X, out_S \setminus X, int_S \cup X, \dashrightarrow_S, \longrightarrow_S)$.

This concludes the introduction of modal I/O automata as well as the strong and weak interface theories. We shall come back to these topics in chapter 5.

2.6 Technical Background

This thesis introduces a variety of tools for supporting the model-driven development of service-oriented software. These tools have been created using different frameworks and libraries, which have proven invaluable to the realisation of the practical part of this thesis, and are shortly introduced in the following.

2.6.1 Eclipse Platform

Nearly all tools presented in this thesis are based on the Eclipse framework [Ecl10i, DFK04]. Eclipse itself is an open source community focused on building tools for software development and thus includes a variety of libraries and frameworks; the Eclipse project was created in 2001 by IBM and taken under the umbrella of the Eclipse Foundation in 2004.

The core of the Eclipse project is an extensible application framework which is able to host all kinds of Java-based desktop and server applications, providing a rich editor- and view-based UI, plug-in support, automatic updates, and various platform services such as a selection service and (retargetable) actions. On top of this framework, a variety of applications have been built; the most well-known of which is the Eclipse Java IDE.

Since version 3.0, the Eclipse platform is based on the OSGi Dynamic Module System for Java [OSG08], which introduces a service layer to the Java programming language. OSGi is based on *bundles*, which are components grouping a set of Java classes and meta-data providing among other things name, description, version, and imported as well as exported packages of the bundle. A bundle may provide arbitrary services to the platform. Each bundle may be dynamically started and stopped as well as added and removed from the platform.

Eclipse provides its own implementation of the OSGi standard named Equinox [GHM+05], and uses the term *plug-in* for bundles integrating into Eclipse. Equinox adds the ability of declarative service descriptions by means of extensions and extension points: Using an XML dialect, a plug-in may describe at which points it may be extended by other plug-ins, and which extensions it contributes to the platform itself. This mechanism enables Eclipse applications to be extended in a flexible way.

OSGi may also be extended by the distributed middleware platform R-OSGi [RAR07]. R-OSGi adds distributed module management to OSGi: Modules may not only be loaded locally, but be deployed, started and stopped on remote machines as well.

Eclipse provides its own library to replace the Java standard Swing/AWT for its user interface. The Standard Widget Toolkit (SWT) [Ecl10f, NW04] is closely based on platform APIs, such that it does not only provide a native *look-and-feel*, but actually uses the widgets available on the platform. SWT is available for various platforms including Windows, Mac OS, and Linux.

There are several libraries available which extend SWT to provide support for different UI use cases such as displaying structured data in tables and trees and for drawing graphical elements by hand. In this thesis, the Draw2D framework[2] is used for the implementation of graphical editors. Draw2D provides a lightweight toolkit of graphical components called *figures*, which can be assembled on a *canvas*. Using *connections*, edges can be drawn between the figures which are automatically routed according to certain constraints.

2.6.2 Model Development Tools and Meta-Models

An important part of this thesis deals with model transformations, which are based on meta-models of certain domains. To create a meta-model, a meta-modelling language is required. In this thesis, the Eclipse Modeling Framework (EMF) [Ecl10a] has been used for this purpose. EMF and MOF are two similar meta-modelling frameworks, which can in fact be translated into one another [GR03]. However, EMF includes the ability to generate a complete Java implementation of a meta-model, which greatly facilitates development and usage of such models.

The preferred way of creating a meta-model in EMF is using an XMI format, for which editors are available in the tool support for EMF. From this format, code can be generated which includes Java classes for all meta-modelling concepts, and a factory for creating concrete instances of the meta-model. Furthermore, EMF provides standard serialisation and deserialisation support for all created models.

As we shall see in later chapters, this thesis introduces two new meta-models for the domains of modal I/O automata and behavioural service specifications. It also re-uses a set of existing meta-models for industry standards, for which an EMF version has already been created.

The first of these is the UML2 meta-model, which is provided by the UML2 subproject of the Eclipse MDT project [Ecl10c]. UML2 is an EMF-based implementation of the Unified Modeling Language (UML) 2 OMG meta-model for the Eclipse platform. As discussed in section 2.2.3, the serialisation and dese-

[2]Draw2D is part of the Graphical Editing Framework (GEF) [Ecl10b].

rialisation support of this project is now in use by many UML2 modellers and thus enables exchange of UML models between these tools.

Another set of meta-models provided by the Eclipse project are the meta-models for the Web Service standards family. As indicated in section 2.1.2, there are numerous Web Service related standards; in this thesis, we are mainly concerned with BPEL, WSDL, SOAP, and XML Schema.

A meta-model for BPEL is provided by the Eclipse BPEL project [Ecl10g]. The meta-model covers the BPEL 2 specification by OASIS, and includes the WSDL extensions for partner link types and message properties for correlation. The meta-models for WSDL and SOAP are provided by the Eclipse Web Tools Platform project [Ecl10j], and covers WSDL 1.1 as well as SOAP 1.2. Lastly, the meta-model for XML Schema is provided by the Eclipse Modelling Tools Project [Ecl10c], which covers XSD version 1.0 provided by the W3C.

Finally, this thesis includes a model transformation to the Java programming language. A meta-model for Java is provided by the Eclipse Model Discovery component (MoDisco) [Ecl10e]. This model is a reflection of the Java language as defined in version 3 of the Java Language Specification [GJSB05] from Sun Microsystems (now Oracle).

2.6.3 Other Tools and Libraries

Three other tools and libraries are worth mentioning as they have been used in the development of the tools in this thesis.

The first of these tools is ANTLR [Par07], a parser generator for Java which has been used to define and implement the data manipulation language introduced in chapter 3.

While creating artefacts for the Web Service family of standards has already been discussed in the last section, executing and testing them is another matter. For this purpose, two tools from the Apache project as well as one commercial tool have been used.

Firstly, the BPEL container Apache ODE [Apa10b] has been used to execute and test the generated BPEL files. Secondly, Apache Axis [Apa10a] along with the Apache Tomcat [Apa10c] server has been used to implement the partners of the generated BPEL processes. Finally, SoapUI [Evi10] has been used to test-drive the running services against their specification.

Chapter 3

Modelling Service Behaviour in UML

The Unified Modeling Language (UML) [OMG10b] is a well-known and mature language for modelling software systems with support ranging from requirement modelling to structural overviews of a system down to behavioural specifications of individual components. However, UML has been designed with object-oriented systems in mind, thus native support and top-level constructs for service-oriented computing such as participants in a SOA, modelling service communication, and compensation support are not included. As a consequence, modelling SOA systems with plain UML requires the introduction of technical helper constructs, which degrades usability and readability of the models.

In this chapter, we therefore introduce a UML extension for SOAs — called the *UML4SOA profile* — which is a high-level domain-specific language for modelling the *behaviour* of services, service orchestrations, and service protocols. For modelling the structural aspects of services, we build on the upcoming OMG standard *SoaML* [OMG09b]. One of the main goals of UML4SOA is minimalism and conciseness: service engineers should have to provide only as much information as necessary for the generation of code, and at the same time as little as possible in order to keep diagrams readable.

In the following, we first introduce the UML4SOA design considerations and the relationship to structural modelling with SoaML (section 3.1). The profile itself is discussed in section 3.2. An example for modelling with UML4SOA is given in section 3.3. Finally, we introduce tool support in section 3.4, discuss related work (section 3.5) and conclude in section 3.6.

Published results: Results presented in this chapter are based on publications [KMH$^+$07], [MSK08b], [MSK08a], [FGK$^+$10a], and [GGK$^+$10]. Furthermore, UML4SOA is a result developed as an answer to one of the main Sensoria research objectives (service-oriented extensions of UML) and has been reported in several technical reports, brochures, and presented at fairs.

3.1 Extending UML for Service Behaviour

The Unified Modeling Language Infrastructure [OMG10a] describes several ways of extending the UML for specific modelling purposes, among them being *profiles*, a lightweight mechanism of defining domain-specific modelling languages on top of the UML. We have introduced the notion of profiles in chapter 2, along with a profile for modelling the *structural* aspects of SOAs (SoaML) which is on the way to becoming an OMG standard.

In this section, we describe why a profile for *behavioural* SOA modelling on top of the UML is desirable before moving on to the description of the profile in the next section.

Behavioural modelling of services and service orchestrations has several requirements on a (graphical) modelling language; in particular, the following three concepts — first-level citizens of a SOA system — should be supported:

- *Communication and Partners.* Services are inherently based on a networked architecture, i.e. communication between services is a key requirement for a working SOA-based system. Communication primitives for sending and receiving calls must thus be supported in a domain-specific SOA language; furthermore, specification of communication partners should be possible in a straightforward way.

- *Long-Running Transactions.* A service, and in particular a service orchestration may represent a business transaction and thus potentially run for a long time. This has various requirements for the modelling language: It must be possible to query the transaction for status updates; it must be possible to handle problems occurring during the transactions, and finally successfully completed transactions might need to be undone due to later failures.

- *Self-Descriptions.* Part of the appeal of service-oriented computing is the focus on a clear self-description of each component in the SOA. Regarding the behaviour, a specification of the *protocol* a service provides or requires is key to enabling quick and confident assembly of SOAs.

UML already includes several ways of specifying the behaviour of software systems which we can extend for modelling service behaviour. In particular, the following two model elements and accompanying diagram types are a good match for modelling SOA behaviour:

- *Activities.* In UML, activities are used for modelling the behaviour of a software component based on a workflow-like paradigm. Workflows are a concept which is also common outside of software modelling; in particular, an interesting area are business processes as they closely match the abstraction level of SOAs. We use activities as the basic mechanism for specifying service and service orchestration behaviour.

- *State Charts.* The UML distinguishes between behavioural and protocol state machines; the first being used to model element behaviour, the second for describing the behaviour of a protocol. As communication is a key aspect of services and service orchestrations, modelling the protocol of a service is important to be able to identify matching service implementations. We therefore use protocol state machines, with a minimal extension to be able to model observed operation calls, in addition to activity modelling for specifying the required or provided protocol of a service.

Attempting to model services using these two UML elements and diagrams while considering the three key aspects of SOA systems discussed above shows several important shortcomings of the UML. We consider our case study as an example for these problems. The structural aspects of the case study, modelled using UML and SoaML, have already been shown in figure 2.9 on page 35.

There are several entities in figure 2.9 for which we might want to specify behaviour. First of all, there is the central participant `ThesisManagement`, which is a service orchestration for which the workflow might be specified. Secondly, the participant contains service and request ports at which services are provided and required, their interfaces being specified as ≪*ServiceInterface*≫s. For these interfaces, the protocol may be specified as a state chart. Finally, the services required by the orchestration have their own behaviour which may be implemented by external means, or may be modelled using UML or UML4SOA as well.

As an example for the three requirements of modelling SOA systems given above, we model the behaviour of the participant `ThesisManagement` and its required and provided protocols. This example will also be used in the remainder of this chapter.

3.1.1 Communication Actions

The first requirement is the ability to specify communications in-between services. The `ThesisManagement` orchestration, for example, is contacted by a student accepting a thesis, subsequently registering this thesis with the examination office. Modelling this sequence as an activity looks like the diagram shown in figure 3.1.

The first action in the figure is an UML `AcceptEventAction` or subclass thereof. It identifies a point in the workflow where the process waits for an incoming event or operation call. The result of the call is placed in an output pin, which is denoted with an arrow leading away from the action. The second action in the figure is an UML `InvocationAction` or subclass, and shows that the process sends out an event or operation call. There are several problems with modelling service communication in this style:

- In UML, there is no graphical distinction between the various subclasses of `AcceptEventAction`, and — even worse — no distinction between an `InvocationAction` and a generic action. This requires a description such

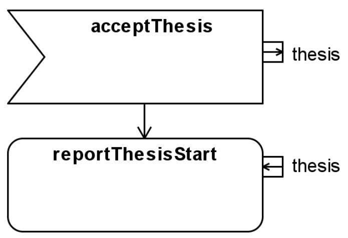


Figure 3.1: eUniversity Case Study: Communications

as the one given above to precisely define the semantics of the diagram. A better way would be to use precisely specified symbols or tags to identify a receiving, sending, or replying action in the sense of a service communication.

- Each of the operations referred to by an action (and denoted in the body) are part of a `ServiceInterface` attached to a service- or request port, specifying the port on which an event or operation is expected or sent out. This way of specifying a port is rather indirect and not visible in the diagram, which does not lay well with the fact that partner services are a first-level concept in a SOA. Furthermore, there is no way of specifying *which* port is to be used if several ports share one `ServiceInterface`. Thus, specifying the port as part of an action would greatly increase readability of the model and also allow for more precise specifications.

- Finally, standard input- and output-pins are used to denote data to be received or sent. In the case of an `AcceptEventAction`, an output pin is used as the data is a result of the action; in the case of an `InvocationAction`, an input is used as the data is used in the action. When considering service calls, however, another intuition is possible: Data received in the process is *input* data, and data sent is *output* data. A different notation for data sent and received in a service context can clarify the intuition in use.

We shall come back to these three problems in the next section, where we define our UML profile.

3.1.2 Long-Running Transactions

The second requirement discussed above is support for *long-running transactions*. Services and, in particular, service orchestrations may be used to specify business or technical processes which potentially run a long time. There should be specific support for such processes in the modelling language; in particular, a long-running transaction may run into problems which need to be handled; it may need to be queried for status updates, and it may need to be rolled back in case of subsequent errors.

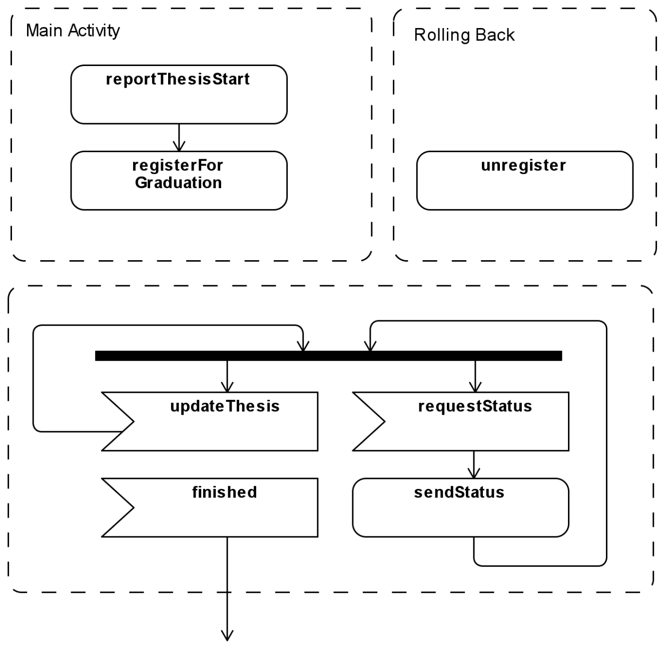


Figure 3.2: eUniversity Case Study: Long-Running Transactions

Consider figure 3.2 which contains a plain UML activity diagram which models two different part of the eUniversity case study. In the upper part, the bootstrapping of the thesis is modelled, in which a student has already accepted a thesis topic and the necessary messages are sent out to register this information. This initialisation might later need to be rolled back (compensated), for example if the student has already been registered for a graduation ceremony. In the lower part, the thesis is in progress. The student provides updates until he is finished, and additionally — and concurrently — the tutor might ask for the current status. Again, there are several problems involved in the figure.

- Firstly, the behaviour for rolling back the main activity later cannot be attached to the activity itself — it must be placed at the point where the rollback(s) takes place and therefore a different place than expected in the diagram. A better way would be associating rollback actions directly with the element to be undone.
- Secondly, modelling concurrent behaviour which might occur multiple times — such as the tutor requesting the status — is difficult to model

in UML. The status requests are in fact optional, which is enabled by the interrupting edge leaving the interruptible activity region. Needless to say, this is not very intuitive to write and read; a better separation between the *main* behaviour and events such as the status updates might be appropriate.

UML4SOA addresses these concerns as discussed in the next section.

3.1.3 Self-Descriptions

Most of the basic definitions of services include a notion of self-description, i.e. the ability of a service to describe, more or less completely, how it can be invoked. The SoaML model for the eUniversity case study shown in figure 2.9 (page 35) already addresses the static aspect of such self-description: The main participant provides two services through the ≪*Service*≫ ports whose operations are given as part of the `StudentServiceInterface` and the `TutorServiceInterface`, and requires two services through its ≪*Request*≫ ports whose operations are given in their respective interfaces. While these descriptions are required as the basis for service interactions, the actual protocol of a provided or required service is not yet specified.

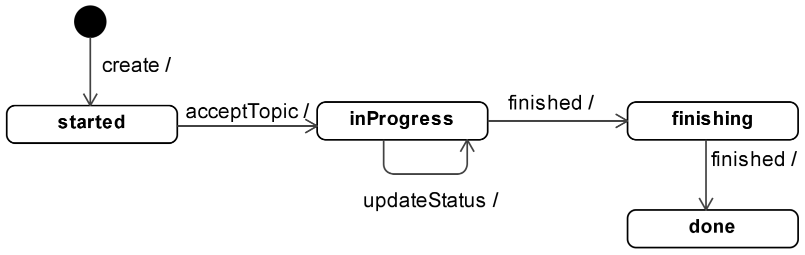


Figure 3.3: eUniversity Case Study: Protocol

UML Protocol State Machines (PrSMs) can be used for this purpose; in our case, they are attached to the types of the SoaML service and request ports, i.e., the `ServiceInterface`s. A protocol state machine in UML may contain states and (protocol) transitions; the latter of which may contain triggers. An example for a standard UML protocol state machine for the `Tutor ServiceInterface` is shown in figure 3.3.

In this figure, two types of events are used: `acceptTopic`, `updateStatus` and the first `finished` transition are based on `ReceiveOperationEvent` triggers, while the second `finished` transition is based on a `SendOperationEvent` trigger. There are two problems associated with this diagram.

First, the fact that the first three transitions actually use a `ReceiveOperationEvent`, and the last an `SendOperationEvent` is not visible in the figure, degrading readability and hampering the comparison with the actual service behaviour (which needs to fulfil this protocol).

Second, the `finished` transition is not legal in the standard definition of protocol state machines. Although it only *observes* an event, this event is not targeted at an operation *implemented* by the classifier the PrSM is attached to, but rather by a required interface of the classifier. However, as we believe that the observation of calls to external services are an important aspect of service specifications, this ability should be added.

Finally, in the interest of ease of modelling and readability, it is again desirable to have a special notation for service-related communication.

Due to the shortcomings discussed above, modelling service behaviour, service orchestrations and protocols with plain UML is a cumbersome task. At the same time, the resulting UML models are difficult to read and translate into executable code. For this reason, we have developed the UML4SOA profile and meta-model which adds specific support for services, service orchestrations and service protocols to the UML.

3.2 The UML4SOA Profile

This section introduces the *UML4SOA* profile, a domain-specific, graphical notation for modelling service behaviour and service protocols. As outlined in chapter 2, extending the UML is possible via several mechanisms, among them MOF meta-model extensions and UML profiles. Both mechanisms can also be combined, an approach which has been used for UML4SOA as well.

In section 3.2.1, we discuss the design decisions behind the profile, introducing — on a high level — the concepts UML4SOA contributes to the UML. In section 3.2.2, we define the meta-model in a MOF modelling style. Mapping of this meta-model to a UML profile takes place in section 3.2.3. Data handling in UML4SOA is discussed in 3.2.4. We discuss the difference between UML4SOA/Open and /Strict in section 3.2.6, and finally talk about life cycle management in section 3.2.7.

3.2.1 Design Considerations

The design of a meta-model for behavioural service specifications requires specific support for the three concepts *communication and partners*, *long-running transactions*, and *self-descriptions* already introduced in section 3.1, which we shall revisit in this section, discussing how UML4SOA addresses these issues.

Furthermore, an important point in specifying service behaviour is data handling, in particular in service orchestrations: Data must be received, might need to be manipulated, and then passed on. Although the UML defines a set of actions for explicitly dealing with data, a textual DSL for data handling greatly simplifies this task for developers.

We begin with defining special support for service communication in activities in section 3.2.1.1, discuss long-running transactions in section 3.2.1.2, introduce self-descriptions as protocols in section 3.2.1.3, and finally discuss data handling in section 3.2.1.4.

3.2.1.1 Service Interactions and Partners

In section 3.1.1, we have noted that although using subclasses of the UML `InvocationAction` or `AcceptEventAction` actions in activities is the preferred way of modelling communication, using this approach suffers from several problems. As a remedy, UML4SOA adds specific support for the requirements of service communication: Firstly, actions are explicitly marked as *sending*, *receiving*, or *replying*. Secondly, we add specialised pins for specifying input and output data. These pins are stereotyped with new icons which precisely show whether data is sent or received. Finally, the partner service an action relates to is attached to an action in a new pin. An UML4SOA diagram replacing the pure UML diagram from section 3.1.1 is shown in figure 3.4.

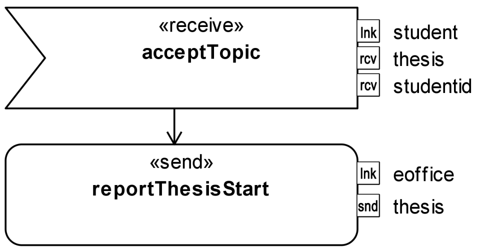


Figure 3.4: eUniversity Case Study: Communications in UML4SOA

As usual in the UML, we employ operation specifications for referencing which functionality is invoked in an interaction. In the case of services, these operations are specified in interfaces or classes used as types of the ≪*Service*≫ or ≪*Request*≫ ports of SoaML. We distinguish:

- service invocations (e.g. `reportThesisStart`), for which we define the new stereotypes ≪*Send*≫ and ≪*Send&Receive*≫ for invoking an action without or with an expected return information. A service invocation is an interaction with a named partner, in which an operation is called, which may, as usual, have parameters and return types.

- service receives (e.g. `acceptTopic`), for which we define the new stereotype ≪*Receive*≫. A service receive is a point where a behaviour waits for an incoming call from a partner, receiving an operation invocation which may, again, have parameters.

- Finally, we add the notion of service replies, which are used to answer a call previously received from a certain partner, and add the new stereotype

≪*Reply*≫ for this notion. As usual in UML, a reply ends a previous invocation.

Each of the service actions may have associated pins which are again stereotyped with UML4SOA stereotypes. They are used to specify the following information:

- a ≪*Lnk*≫ (link) pin specifies the partner for an operation (i.e., the port through which messages are sent or received, for example `student` or `eoffice` in the example above),
- a ≪*Rcv*≫ (receive) pin denotes where received information is stored. In general, this will be a variable (`studentId` and `thesis` in the example above),
- a ≪*Snd*≫ (send) pin denotes the information to be sent as part of a call (the variable contents of the `thesis` variable in the example above). Besides the contents of variables, such data may also be generated on-the-fly (for example, by string concatenation).

More information about the data handling syntax is given in 3.2.1.4.

3.2.1.2 Events and Compensation

Handling long-running transactions in the SOA world has two major requirements. First of all, it should be possible to query a long-running service for its status or other information, which is additional work the service has to carry out in addition to the normal behaviour. Secondly, successfully completed work might need to be undone in a customised way (transaction rollback).

The first issue is handled in UML4SOA by means of *event handlers*, which allow the specification of optional behaviour occurring in parallel to the main work of a service. The second issue is handled by means of *compensation handlers*, which allow attaching roll-back behaviour to a certain action or set of actions of a process definition. An example of both is shown in figure 3.5, the counterpart of figure 3.2 in UML4SOA.

As the figure shows, UML4SOA introduces a grouping concept — the ≪*ServiceActivity*≫ — to which specialised edges for event and compensation handling can be attached.

Firstly, the figure shows how to attach compensation handling to an area in UML4SOA (top half). We use a specialised edge ≪*Compensation*≫, indicating that compensation handling is available for a certain area. In the example, the complete `Registration` activity can be rolled back by executing the action in the `CompensationHandler` activity. By attaching these actions to the area to be compensated, we only need to specify them once, and they are defined in close context to their counterpart.

With regard to events, UML4SOA includes a specialised ≪*Event*≫ edge to attach an event handler to a certain area. In the example, an event handler is

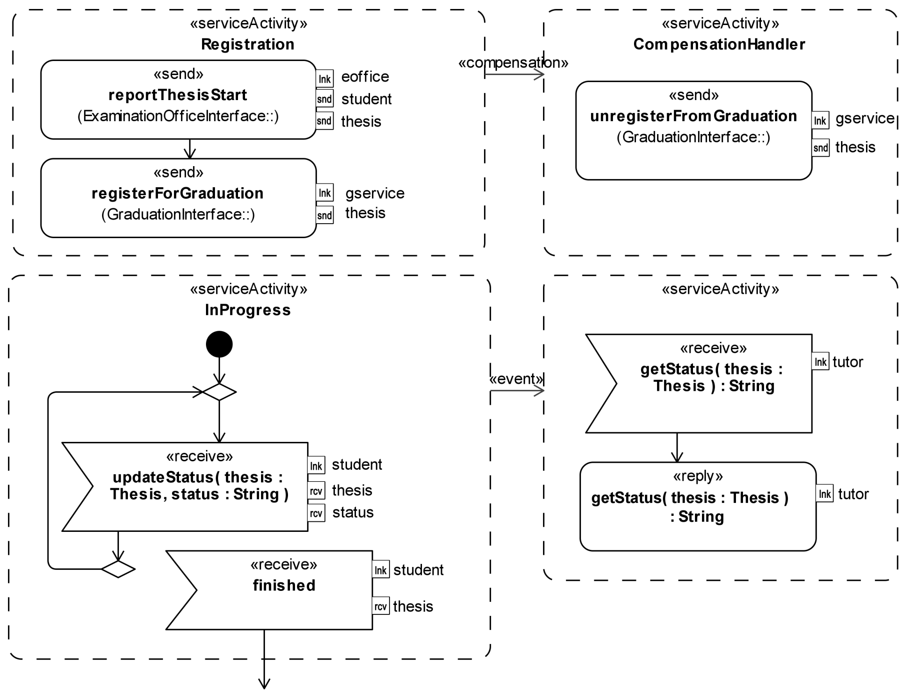


Figure 3.5: eUniversity Case Study: Long-Running Transactions in UML4SOA

attached to the `InProgress` activity. This means that during the waiting time for any number of `updateStatus` calls or a final `finished` call, a `getStatus` call might come in and is answered. In general terms, an event is a message which might be accepted during the run of a certain element in the workflow, asking — for example — for status information or for cancellation. The ≪*Event*≫ edge allows us to specify such events in an easy and readable way.

3.2.1.3 Self-Describing Protocols

UML4SOA orchestration specifications are complemented by protocols assigned to the ports of the SoaML participant. A port protocol always describes actions of the ≪*Participant*≫ — either actions provided to clients or actions invoked on or expected from partners.

As noted in the previous section, the definition of protocol state machines in UML [OMG10b] allows referencing operations *implemented* by the context classifier. An important part of service behaviour, on the other hand, is sending out calls to partner services; these operations are *used* by context classifiers. UML4SOA thus extends the ability of PrSMs to include triggers with a `SendOperationEvent` event. It is important to note that this event is not an *effect* of a transition; rather; it is an *observed operation call* of the participant the classifier of the PrSM is attached to.

As already discussed in section 3.2.1.3, it is again beneficial to tag the individual parts of a PrSM to clarify the semantics of UML4SOA protocols. The following actions may be observed in a service protocol:

- The receipt of an invocation is observed as a (UML) `ReceiveOperationEvent`. UML4SOA adds a special transition with this constraint with the ≪*Receive*≫ stereotype.
- A service invocation is observed as a (UML) `SendOperationEvent`. UML4SOA adds a special transition with this constraint with the ≪*Send*≫ stereotype.
- As noted above, a service invocation might be a reply to a previous ≪*Receive*≫. For clarity, this is modelled separately in UML4SOA, although we observe again a (UML) `SendOperationEvent`. The UML4SOA stereotype for a service reply transition is ≪*Reply*≫.
- Finally, a protocol may also need to specify the fact that a reply is *expected* from a partner in response to a previous ≪*Send*≫. The UML4SOA stereotype for an expected reply transition is ≪*ReceiveReply*≫; the event used is again a `ReceiveOperationEvent`.

A matching UML4SOA diagram for figure 3.1 is shown in figure 3.6.

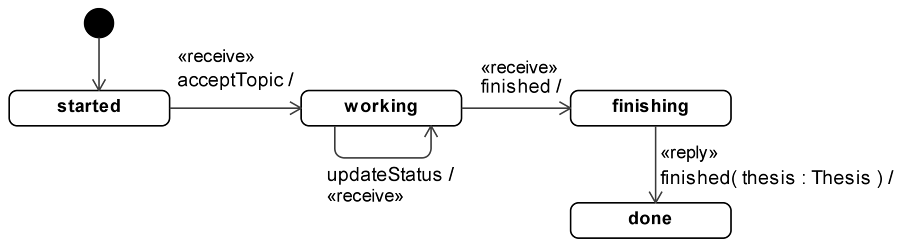


Figure 3.6: eUniversity Case Study: Protocols in UML4SOA

The protocols specified at the ports of a ≪*Participant*≫ must *match* the behaviour of the orchestration. Furthermore, clients and partners must be *compatible* with the given protocol for the system to work correctly. This will be discussed in more detail in chapter 5.

3.2.1.4 Data Handling

Services in a real-world environment are about data: Whether from user input, databases, or as a result of a lengthy calculation, data needs to be handled within services and received via or sent over the network. This holds especially true for service orchestrations, part of whose job is the distribution of data.

The SoaML profile introduced in section 2 already defines the static aspects of data handling with the ≪*MessageType*≫ stereotype, which tags data classes without behaviour to be used in operation calls. In UML activities, certain actions such as `ReadVariableAction` or `AddVariableValueAction` are available for dealing with data. In the interest of both readability and usefulness, UML4-SOA replaces these with a simple data manipulation language on top of message types, which allows assignments of (parts of) data and manipulations such as simple mathematical operations or string concatenation. It is important to note that this language is not an *action language*; its sole purpose is the specification of data manipulation statements.

A syntax for data handling is relevant in three parts of UML4SOA specification:

- *Variables.* (UML) variables hold the data of the service. A variable is referred to in UML4SOA receive pins (for storing data) and in send pins (for data to be sent out).
- *Guards.* A guard (for example on an outgoing edge from a decision node) may contain a boolean condition which might reference data from one of the variables.
- *Explicit Data Operations.* Finally, sometimes using on-the-fly data handling is not enough; for example, when performing complex copy operations between input and output operations. UML4SOA introduces a specific action for these transactions.

The UML4SOA data handling language is a strongly typed language built on primitive data types and the message types defined in the SoaML model part. Inspired by the Java syntax, the language reads like pseudo code and matches the overall abstraction layer of UML4SOA. An example for three different expressions in this language is shown in listing 3.1.

Listing 3.1: UML4SOA Data Handling Example

```
String currencies;
currencies= convert.from + ”-” + convert.to
::Main.conversionInfo= currencies
```

The first line declares a variable `currencies` with the well-known UML type String. The second line assigns a field of a variable `convert` concatenated with the constant string “-” concatenated with another field of `convert` to `currencies`. The third line assigns `currencies` to an existing variable `conversionInfo` which resides in an enclosing element called `Main`.

In general, a UML4SOA ≪*Rcv*≫ pin contains what is usually regarded as the left-hand side of an expression, i.e. the specification of where to store data. This will normally be some variable which, if it does not exist, is implicitly created in UML4SOA. A ≪*Snd*≫ pin may contain what is usually regarded as the right-hand side of an expression, i.e. a complete statement including mathematical or string operations.

Regarding more complex data operations, UML4SOA adds a new action for data handling with the stereotype ≪*Data*≫, which may contain statements for declaring variables and manipulating data.

3.2.2 The UML4SOA Meta-Model

In this section, we define the UML4SOA meta-model which forms the basis for the UML4SOA profile. The meta-model is closely based on the UML meta-model and in particular, UML activities and protocol state machines.

The two figures 3.7 and 3.8 on pages 62 and 63 show the complete meta-model. Grey classes are taken from the UML while white classes are newly defined in UML4SOA. Note that some classes appear twice for layouting purposes.

The two figures can be grouped into four areas. In the first, the top shows the main structuring element of UML4SOA (`ServiceActivityNode`) along with actions and edges for specifying compensation and data. We shall discuss these in section 3.2.2.1. The bottom of this figure shows the protocol extensions, which we discuss last (section 3.2.2.4).

In the second figure, the service communication actions are shown, which are linked to data pins on the bottom. We shall discuss the pins first in section 3.2.2.2, afterwards using them in the definition of the communication actions in section 3.2.2.3.

3.2.2.1 Structuring Elements

Structuring service behaviour diagrams is important not only for readability, but for being able to handle events and compensation in a straightforward way. As discussed in the last section, UML4SOA introduces the structuring concept of `ServiceActivityNodes`, to which event and compensation handlers can be attached. Compensation handlers can later be invoked by using specialised actions.

ServiceActivityNode

Description

A `ServiceActivityNode` represents either

1. a special `Activity` for service behaviour, or
2. a grouping element for actions and other `ServiceActivityNodes` (top-level, and nested)

A `ServiceActivityNode` may have control edges connected to it, and pins when merged with CompleteActivities or on specialisations in CompleteStructuredActivities. The execution of any embedded actions may not begin until

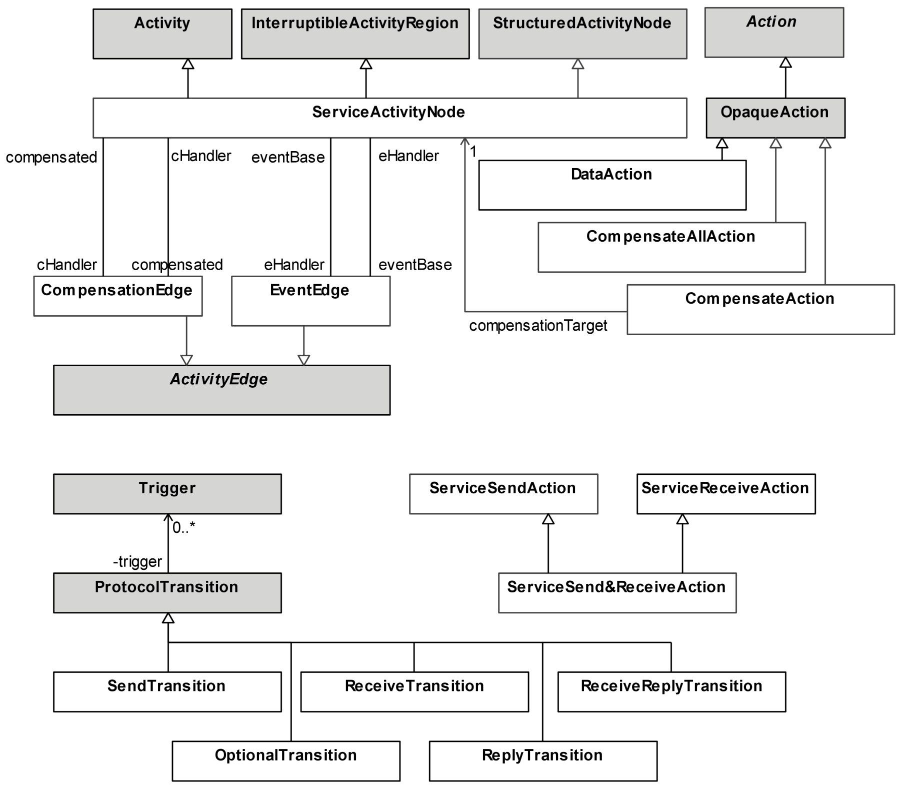


Figure 3.7: UML4SOA Meta-Model (Structures and Protocols)

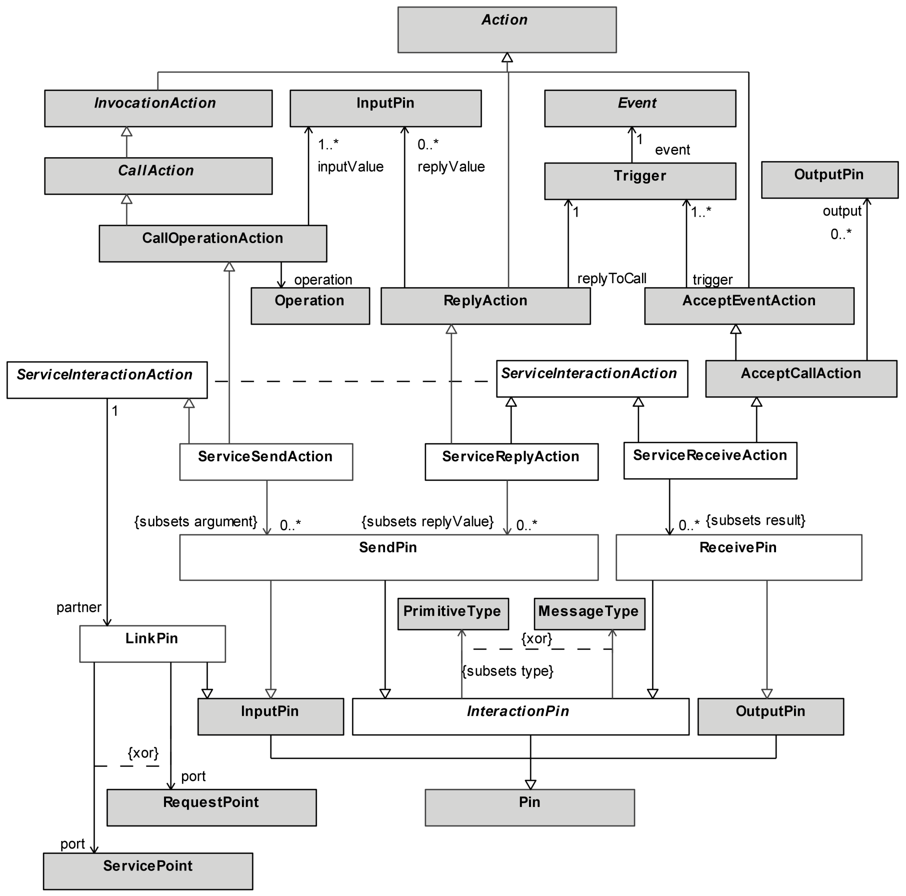


Figure 3.8: UML4SOA Meta-Model (Actions and Pins)

the `ServiceActivityNode` has received its object and control tokens. The availability of output tokens from the structured activity node does not occur until all embedded actions have completed execution. Note that completion waits for already running event handlers.

In addition to both `Activity` and `StructuredActivityNode`, a `ServiceActivityNode` node may have attached event and compensation handlers. An event handler may be executed at any time during the execution of the `ServiceActivityNode`, running in parallel to the `ServiceActivityNode`. Event handlers may be invoked multiple times. A compensation handler, on the other hand, defines behaviour to be executed to undo the work of a successfully completed `ServiceActivityNode`. Note that if no compensation handler is defined for a `ServiceActivityNode`, a default handler with a ≪*CompensateAll*≫ action is assumed.

Furthermore, interrupting edges may halt execution at any time (as already defined in the UML class `InterruptibleActivityRegion`).

A top-level service activity is attached to a SoaML ≪*Participant*≫ and defines the behaviour of the ≪*Participant*≫ across all service- and request ports.

Generalisations

- `StructuredActivityNode`
- `InterruptibleActivityRegion`
- `Activity`

Associations

- `eHandler` : `EventEdge`[0..*]
  An event edge leading to an event handler for this activity.
  {subsets `outgoing`}

- `eventBase` : `EventEdge`[0..*]
  An event edge leading to another activity for which this activity is an event handler.
  {subsets `incoming`}

- `cHandler` : `CompensationEdge`[0..1]
  A compensation edge leading to a compensation handler for this activity.
  {subsets `outgoing`}

- `compensated` : `CompensationEdge`[0..1]
  A compensation edge leading to another activity for which this activity is the compensation handler.
  {subsets `incoming`}

Constraints

1. If a compensation handler is specified, the target element must have this element as the `compensated` element.

2. If event handlers are specified, each of them must have this element as the `eventBase` element.

Notation

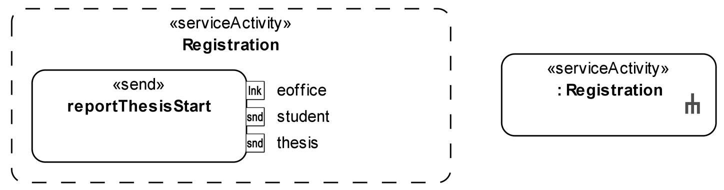
As a `ServiceActivityNode` comes in two versions, there are two notations.

1. `StructuredActivityNode` notation: the `ServiceActivityNode` is drawn with a dashed round cornered rectangle enclosing its nodes and edges, with the stereotype notation ≪*ServiceActivity*≫ at the top. Also see children of `StructuredActivityNode`.

2. `Activity` notation: Same notation as for activities applies; as before, the ≪*ServiceActivity*≫ stereotype must be used.

Examples

The following examples show the use of a service activity. The activity on the left contains one action with the stereotype ≪*Send*≫, which in turn contains three pins. The service activity is annotated with the ≪*ServiceActivity*≫ stereotype, and carries a name (`Registration`). On the right, an action for invoking the activity without displaying the internals is shown.



CompensationEdge

Description

A `CompensationEdge` is an edge connecting a `ServiceActivityNode` to be compensated with the one specifying a compensation. It does not model a normal control flow — instead, it indicates an association between a (main) service element and a compensation handler. Execution of a compensation handler is triggered with a `CompensateAction` or a `CompensateAllAction`.

Exceptions thrown during a compensation handler must be handled in the invoking `ServiceActivityNode`, or in a handler attached to the compensation handler.

Generalisations

- `ActivityEdge`

Associations

- `compensated` : `ServiceActivityNode`[1..1]
  The service activity which is compensated.
  {subsets `source`}

- `cHandler` : `ServiceActivityNode`[1..1]
  The service activity specifying the compensation actions.
  {subsets `target`}

Constraints

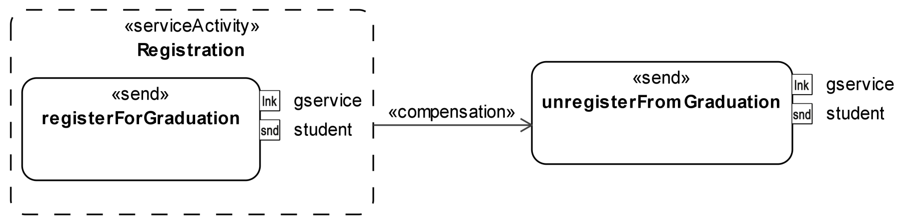
The compensated element must have this element as the compensation handler. A compensation edge may only be attached to a service activity.

Notation

The edge is annotated with the stereotype ≪*Compensation*≫.

Examples

This example shows the use of a ≪*ServiceActivity*≫-typed compensation handler. An ordinary ≪*ServiceActivity*≫ registers a student for a graduation celebration event. Later on, if the student fails to graduate, the compensation handler is invoked to unregister the student.



EventEdge

Description

An `EventEdge` is an edge connecting event handlers with a `ServiceActivityNode` during which the event may occur. It does not model a normal control flow — instead, it indicates an association between a (main) service element and an event handler.

Execution of an event handler is triggered externally by means of a call, or a timed event. An event handler may be executed zero, one, or multiple times in parallel to the service element it is attached to.

Note that only one instance of a specified event handler is active at the same time.

Generalisations

- `ActivityEdge`

Associations

- `eventBase` : `ServiceActivityNode`[1..1]
  The service activity to which an event handler is attached.
  {subsets `source`}

- `eHandler` : `ServiceActivityNode`[1..1]
  The service activity specifying an event handler.
  {subsets `target`}

Constraints

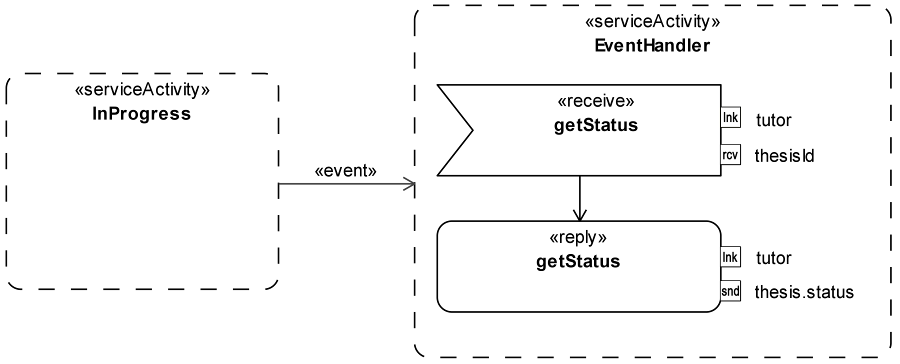
The event base element must have this element as an event handler. An event edge may only be attached to a service activity.

Notation

The edge is annotated with the stereotype ≪*Event*≫.

Examples

This example shows the use of a ≪*ServiceActivity*≫-typed event handler. The event handler is installed in parallel to the `InProgress` activity, and allows to retrieve the status of the service with a call (`getStatus`). This happens in parallel to the `InProgress` activity.



CompensateAction

Description

The `CompensateAction` invokes the compensation handler for a particular `ServiceActivityNode`, whose name is given in the body of the action and which must be nested inside the service element the handler in which the `CompensateAction` is specified in is attached to.

A `CompensateAction` may only be invoked from an exception or compensation handler. After the compensation handler of the given `ServiceActivityNode` has been executed, the instance is removed (*uninstalled*) from the referenced node, and the execution resumes normally after the `CompensateAction`.

Generalisations

- `OpaqueAction`

Associations

- `compensationTarget` : `ServiceActivityNode`[1..1]
  The `ServiceActivityNode` to be compensated.

Constraints

- The `CompensateAction` may only be used within a compensation or exception handler.

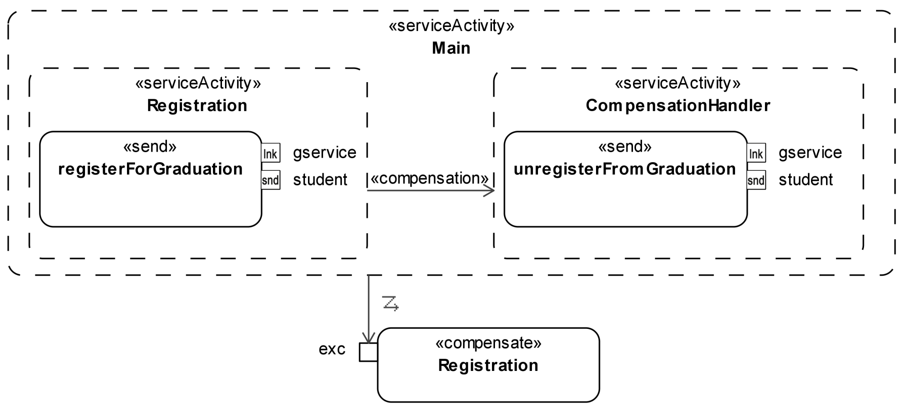
- The `compensationTarget` must be a `ServiceActivityNode` which has a compensation handler, and that `ServiceActivityNode` must be nested within the `ServiceActivityNode` in which the compensation action is invoked.

Notation

Annotation with stereotype ≪*Compensate*≫. The target name is given inside the body of the action.

Examples

This example shows the use of the compensate action. In this example, the compensation handler of `Registration` is invoked by means of a ≪*Compensate*≫ action.



CompensateAllAction

Description

The `CompensateAllAction` invokes all installed compensation handlers which are nested in the `ServiceActivityNode` to which the handler the `Compensate-AllAction` is specified in is attached to.

A `CompensateAllAction` may only be invoked from an exception or compensation handler. It starts compensation of all inner `ServiceActivityNodes` of the `ServiceActivityNode` the exception- or compensation handler the action is defined in is attached to.

The inner `ServiceActivityNode`s with compensation handlers are compensated in reverse order of their completion, i.e. the last completed `ServiceActiv-ityNode` first. However, this applies only if the `ServiceActivityNode`s are on the same level; inside the compensation handlers which are started in reverse order, the inner compensated `ServiceActivityNode`s compensation handlers might not necessarily run in (global) reverse order (they do in local reverse order).

After the compensation handlers have been executed, the instances are removed (*uninstalled*) from their respective `ServiceActivityNode`s, and the execution resumes normally after the `CompensateAllAction`.

Generalisations

- `OpaqueAction`

Associations

None.

Constraints

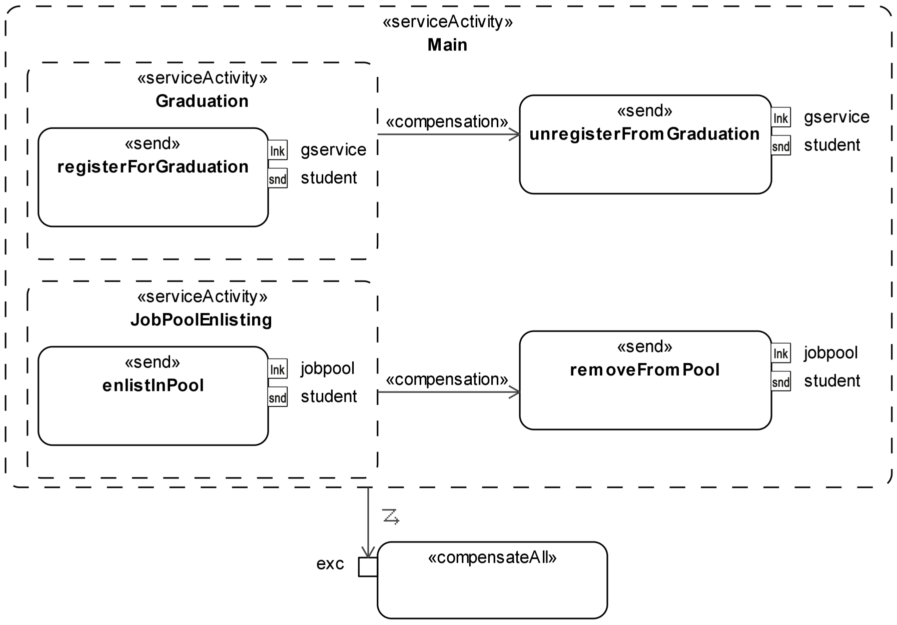
The `CompensateAllAction` may only be used within a compensation or exception handler.

Notation

Annotation with stereotype ≪*CompensateAll*≫.

Examples

In this example, two service activities are present. Each has an attached compensation handler. The first is installed after the `Graduation` activity completes; the second after `JobPoolEnlisting`. Both can potentially be invoked with the ≪*CompensateAll*≫ call if an exception is caught in the `Main` scope.



DataAction

Description

A `DataAction` is an action for data manipulation, for example, declaring variables and manipulating them (assignments, calculations, etc.). The `DataAction` allows the specification of arbitrarily many statements, written in the domain-specific UML4SOA expression language (see Sect. 3.2.4).

Generalisations

- `OpaqueAction`

Associations

None.

Constraints

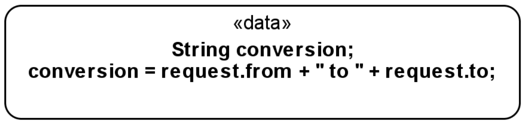
No additional constraints.

Notation

A `DataAction` is stereotyped with ≪*Data*≫. The statements to be executed are given inside the body.

Examples

This example shows a data action. In the action, a string-typed variable is declared (`conversion`). Afterwards, conversion is assigned by using two fields of the `request` variable, a string (“ `to` ”), and the string concatenation operator “+”.



3.2.2.2 Pins

This section lists the pin classes of UML4SOA, which are used in service interactions for denoting partners as well as received and sent calls. The pin meta-classes are shown at the bottom of figure 3.8.

LinkPin

Description

A `LinkPin` is used to indicate the partner service for the service interaction. As a partner service is indicated through the ports of the participant to which the main `ServiceActivityNode` is attached to, the `LinkPin` is bound to a port. At runtime, an instance of the port is dynamically provided at `LinkPin`s.

Note that for `LinkPin`s referencing `Request` ports, a partner must be bound before execution by external means. For `Service` ports, incoming calls trigger creation of a new port instance which is given in the `LinkPin`.

Generalisations

- `InputPin`

Associations

- `port` : `Service` *xor* `Request`[1..1]

Constraints

- The `port` must be attached to the class which the root `ServiceActivity-Node` of this behavioural specification belongs to.

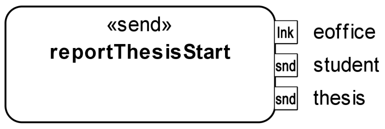
Notation

The pin is stereotyped with ≪*Lnk*≫, or with the corresponding icon ("lnk"). The port name is specified along with the pin.

Examples

This example shows the use of a `LinkPin`. In all partner-related actions, for example in this ≪*Send*≫, the port at which the operation is requested or received must be specified.

In the example, the port is a ≪*Service*≫ port of the corresponding participant which carries the name `eoffice`.



InteractionPin

Description

An `InteractionPin` serves as the common abstract base class of `SendPin` and `ReceivePin`, restricting their type to either `MessageType` or `PrimitiveType`.

Generalisations

- `Pin`

Associations

- (inherited association from supertype) : `MessageType` *xor* `PrimitiveType`[1..1]
  {subsets `type`}

Constraints

The type must be a subtype of either `MessageType` or `PrimitiveType`.

Notation

None.

SendPin

Description

A `SendPin` is used in send actions to denote the data to be sent to an external service. A `SendPin` specifies data to be transmitted. Arbitrary right-hand side expressions specified in the UML4SOA expression language may be used.

Generalisations

- `InputPin`
- `InteractionPin`

Associations

No additional associations.

Constraints

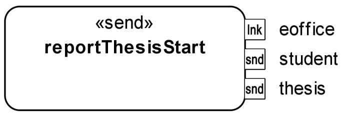
The type must be a subtype of either `MessageType` or `PrimitiveType`. Also, the `SendPin` must have the correct type for the operation and partner invoked.

Notation

The `SendPin` must be stereotyped with ≪*Snd*≫, or with the corresponding icon ("snd"). Furthermore, it needs to be annotated with the information about data to be sent. In UML, pins are ordered, which cannot directly be shown in the diagram. As a convention, UML4SOA send pins should be denoted on the right-hand side of an action from top to bottom.

Examples

This example shows the use of two `SendPin`s; this means that the operation used (`reportThesisStart`) requires two parameters. The first send pin specifies the variable `student`; the second the variable `thesis`.



ReceivePin

Description

A `ReceivePin` is used in receive actions to denote the place where the data received from an external service is stored (i.e., a variable, or a part of a variable).

Generalisations

- `OutputPin`
- `InteractionPin`

Associations

No additional associations.

Constraints

The type must be a subtype of either `MessageType` or `PrimitiveType`. Also, the `ReceivePin` must have the correct type for the operation and partner invoked.

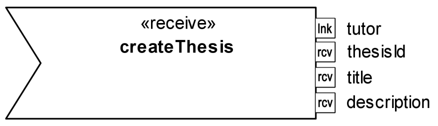
Notation

The `ReceivePin` must be stereotyped with ≪*Rcv*≫, or with the corresponding icon ("rcv"). Furthermore, it needs to be annotated with the information about where to store the received data.

In UML, pins are ordered, which cannot directly be shown in the diagram. As a convention, UML4SOA send pins should be denoted on the right-hand side of an action from top to bottom.

Examples

This example shows the use of `ReceivePins`. There are three receive pins; each for one of the parameters of the `createThesis` call. Each pin contains the target where the data will be stored; in this case, these are all variable names.



3.2.2.3 Communication Actions

Having introduced structuring elements and pins for data handling, we can now discuss the specialised actions for communication in UML4SOA. These actions are displayed on the top of figure 3.8.

ServiceInteractionAction

Description

ServiceInteractionAction is the common base class of all service interaction actions which have an associated LinkPin. The interaction is linked to a partner (i.e. a certain port) of the behaviour via the link pin (see section 3.2.2.2 for more information on LinkPins). The operation is specified in the actions themselves.

Generalisations

None.

Associations

- partner : LinkPin[1..1]
  Specifies the partner of this ServiceInteractionAction. In case of a ServiceSendAction, this association subsets target.

Constraints

No additional constraints.

Notation

No notation.

ServiceSendAction

Description

A ServiceSendAction is an action that invokes an operation of a target service without expecting a return value. The argument values are data to be transmitted as parameters of the operation call. CallOperationAction contains the operation directly.

ServiceSendAction inherits argument from InvocationAction. We restrict this to SendPins which contain the data to be sent.

Generalisations

- CallOperationAction
- ServiceInteractionAction

Associations

- (inherited association from supertype) : SendPin[0..*]
  {subsets argument}

Constraints

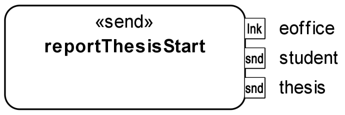
- `ServiceSendAction` constrains `argument` (inherited from `InvocationAction`) to pins of type `SendPin`.
- `target` is constrained to instances of `LinkPin`.

Notation

A `ServiceSendAction` is stereotyped with ≪*Send*≫. The operation name is given inside the action body.

Examples

This example shows a send. An operation call is sent to the partner attached to the port `eoffice` (specified in the link pin). The data to be sent is stored in two variables: `student` and `thesis` (specified in the send pins). There is no return value.



ServiceReceiveAction

Description

A `ServiceReceiveAction` is an accept call action representing the receipt of an operation call from an external partner. No answer is given to the external partner.

A `ServiceReceiveAction` blocks until the specified operation call is received. It requires a trigger (with a `CallEvent` event), which contains the operation. According to the operation, appropriate `ReceivePin`s must be given which contain the variables in which the incoming data is stored.

Note that there is a caveat involved with attaching, through the superclass `ServiceInteractionAction`, a `LinkPin` to a `ServiceReceiveAction`. The former is an `InputPin`, while the second is an `AcceptCallAction`. Unfortunately, the UML superstructure defines a constraint on `AcceptEventAction` (the direct superclass of `AcceptCallAction`, prohibiting the use of `InputPin`s on this class. This will be further discussed in section 3.2.5.

Generalisations

- `ServiceInteractionAction`
- `AcceptCallAction`

Associations

- (inherited association from supertype) : `ReceivePin`[0..*]
  {subsets `result`}

Constraints

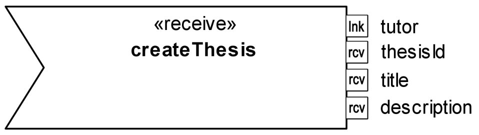
The result pins must be `ReceivePin`s. This ensures that the data received has value or message types. The trigger must be a `CallEvent`.

Notation

A `ServiceReceiveAction` is stereotyped with ≪*Receive*≫. The operation name (from `trigger` → `CallEvent`) is given inside the action body.

Examples

This example shows a receive. A call is received from a partner (called `tutor`, specified in the link pin). The data is stored in three variables (`thesisId`, `title`, and `description` (specified in the receive pins). The operation invoked is called `createThesis`.



ServiceReplyAction

Description

`ServiceReplyAction` is an action that accepts a return value and a value containing return information produced by a previous `ServiceReceiveAction`. The reply action returns the values to the request port of the previous call, completing execution of the call.

`ServiceReplyAction` is a specialised version of `ReplyAction` for the service-oriented context. The inherited attribute `replyValue` is subset to point to instances of `SendPin`, instead of a generic input pin, thereby ensuring the data can be interpreted as value data. Thus, a `ServiceReplyAction` sends back data to a request port for which previous data was received.

Generalisations

- `ReplyAction`
- `ServiceInteractionAction`

Associations

- (inherited association from supertype) : `SendPin`[0..*]
  {subsets `replyValue`}

Constraints

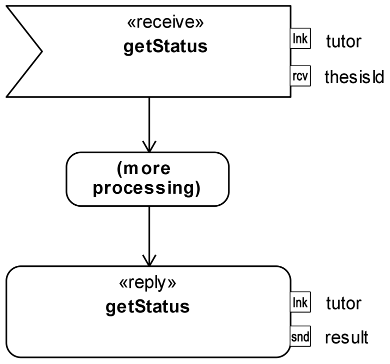
The `replyValue` pins must be of type `SendPin`.

Notation

A `ServiceReplyAction` is stereotyped with ≪*Reply*≫. The operation name is given inside the action body (corresponding to the operation inside the attached trigger).

Examples

This example shows a reply. A reply is always an answer to a previous receive, and carries the same partner and operation name as the receive. In this example, a `getStatus` call is received from partner `tutor`, and the single parameter is stored in the variable `thesisId`. Now, some processing takes place. Afterwards, the data in the variable `result` is sent as a reply to the `tutor` partner.



ServiceSend&ReceiveAction

Description

A `ServiceSend&ReceiveAction` action is a complete operation call execution with a partner. Some data (stored in the `SendPins`) is sent, then the action waits for data to be sent back, which is stored in the `ReceivePins`.

Generalisations

- `ServiceSendAction`
- `ServiceReceiveAction`

Associations

None.

Constraints

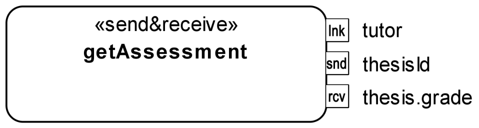
No additional constraints.

Notation

A `ServiceSend&ReceiveAction` is stereotyped with ≪*Send&Receive*≫. The operation name is given inside the action body.

Examples

This example shows a ≪*Send&Receive*≫. An operation is invoked on the partner `tutor` (specified in the link pin). The data itself is stored in the variable `thesisId` (specified in the send pin) and must be initialised before the action. The return value from the service is stored in the element `grade` of the variable `thesis` (specified in the receive pin).



3.2.2.4 Protocols

This section lists specialised transitions for denoting send, receive, and reply operations of a participant a UML4SOA protocol state machines belongs to.

ReceiveTransition

Description

A specialised transition indicating that an operation call is received by the participant to which the protocol state machine is attached to.

Generalisations

- `ProtocolTransition`

Associations

None.

Constraints

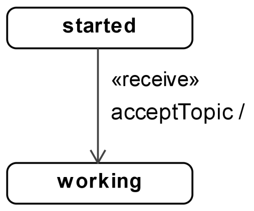
The trigger of this transition must be a `ReceiveOperationEvent`. Furthermore, the event must reference an operation *implemented* in the port type the PrSM is attached to.

Notation

Annotation with stereotype ≪*Receive*≫.

Examples

This example shows a ≪*Receive*≫ in a protocol state machine. The example contains two states, `started` and `working`. In the `started` state, the operation call `acceptTopic` is expected, which leads the PrSM to the `working` state.



SendTransition

Description

A specialised transition indicating that an operation is invoked without returning information by the participant to which the protocol state machine is attached to. The operation invoked must be specified in a required interface of the classifier the protocol state machine is attached to.

Generalisations

- `ProtocolTransition`

Associations

None.

Constraints

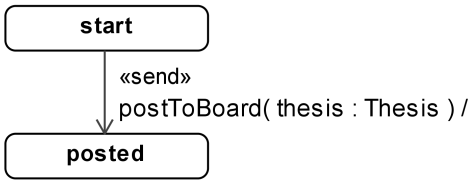
The trigger of this transition must be a `SendOperationEvent`. Furthermore, the event must reference an operation implemented in an interface *used* in the port type the PrSM is attached to.

Notation

Annotation with stereotype ≪*Send*≫.

Examples

This is an example for using a send transition. Two states are used: `start` and `posted`. In the `start` state, the participant may choose to send out the `postToBoard` call; in this case, the PrSM is advanced to the `posted` state.



ReplyTransition

Description

A specialised transition indicating that a previous operation call is being replied to by the participant to which the protocol state machine is attached to.

Generalisations

- `ProtocolTransition`

Associations

None.

Constraints

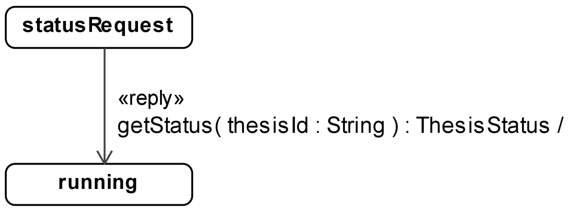
The trigger of this transition must be a `SendOperationEvent`. The event must reference an operation implemented in the port type the PrSM is attached to.

Notation

Annotation with stereotype ≪*Reply*≫.

Examples

This is an example for using a reply transition. At the beginning, the PrSM is in the `statusRequest` state. Here, the participant may choose to reply to the `getStatus` call. The PrSM is advanced to the `running` state.



ReceiveReplyTransition

Description

A specialised transition indicating that an operation call is received by the participant to which the protocol state machine is attached to, and that this receive is in response to a previous send originating from this participant.

Generalisations

- `ProtocolTransition`

Associations

None.

Constraints

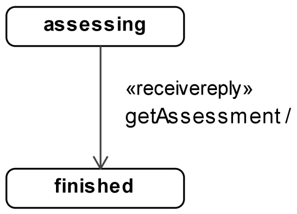
The trigger of this transition must be a `ReceiveOperationEvent`. The event must reference an operation implemented in the port type the PrSM is attached to.

Notation

Annotation with stereotype ≪*ReceiveReply*≫.

Examples

This example shows a ≪*ReceiveReply*≫ in a protocol state machine. The example contains two states, `assessing` and `finished`. In the `assessing` state, a reply to the operation `getAssessment` is expected, which leads the PrSM to the `finished` state.



OptionalTransition

Description

`OptionalTransition` is a specialised transition indicating that the operation given as part of this transition (specified with ≪*Send*≫, ≪*Receive*≫, ≪*Receive-reply*≫ or ≪*Reply*≫) is optional, i.e. may or may not be supported by an implementation of this protocol.

Generalisations

- `ProtocolTransition`

Associations

None.

Constraints

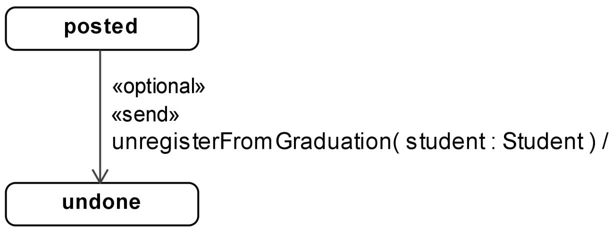
The transition must also be annotated with ≪*Send*≫, ≪*Receive*≫, ≪*Receive-reply*≫, or ≪*Reply*≫.

Notation

Annotation with stereotype ≪*Optional*≫.

Examples

This is an example for using an optional send transition. If the PrSM is in the state `posted`, the participant may choose to send the `unregisterFromGraduation` call, leading to the state `undone`.



3.2.3 From Meta-Model to Profile

As indicated at the beginning of section 3.2, the aim is defining a lightweight extension of the UML in the form of a profile. In the previous section, we have defined a meta-model for UML4SOA; we now map the meta-classes and attributes of this meta-model to stereotypes and tag definitions.

As a UML profile, UML4SOA defines a profile package whose meta-model reference-element is the UML, and which additionally imports the stereotypes from the SoaML profile (see figure 3.9).

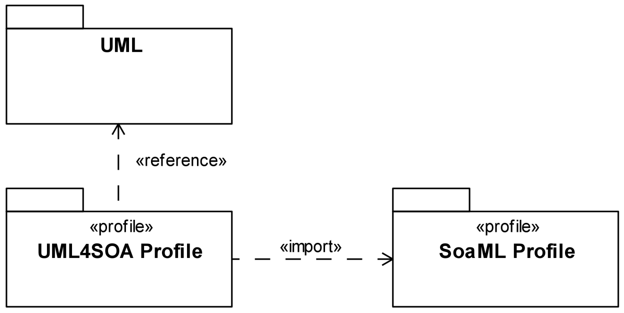


Figure 3.9: UML4SOA Profile Package

The following figures will each show several meta-classes and their mapping to stereotypes. The notation is as follows:

- UML meta-classes, which are extended by the stereotypes of the UML4-SOA profile, are shown in gray.
- Stereotypes of the UML4SOA profile are shown in yellow. Note that these may only *extend* UML meta-classes, which is shown by the extension arrows.
- Finally, as a reference, the UML4SOA meta-classes are shown in white. A ≪*mapsTo*≫ relation between a stereotype and an UML4SOA meta-class gives the intuition of which stereotype represents which meta-class; this notation is not defined in the UML.

We attach a specific semantic meaning to the ≪*mapsTo*≫ relationship: If a stereotype *maps to* a UML4SOA meta-class, it is subject to the same constraints regarding inherited associations. For example, the meta-class `ServiceSend-Action` requires that all arguments must be `SendPins`; this is transferred to the ≪*Send*≫ stereotype.

Note that the stereotypes are *defined* with an uppercase letter; by contrast, the *application* of a stereotype uses a lowercase letter. This style is in line with section 18.3.8 (Stereotypes) of the UML superstructure (see [OMG10b]).

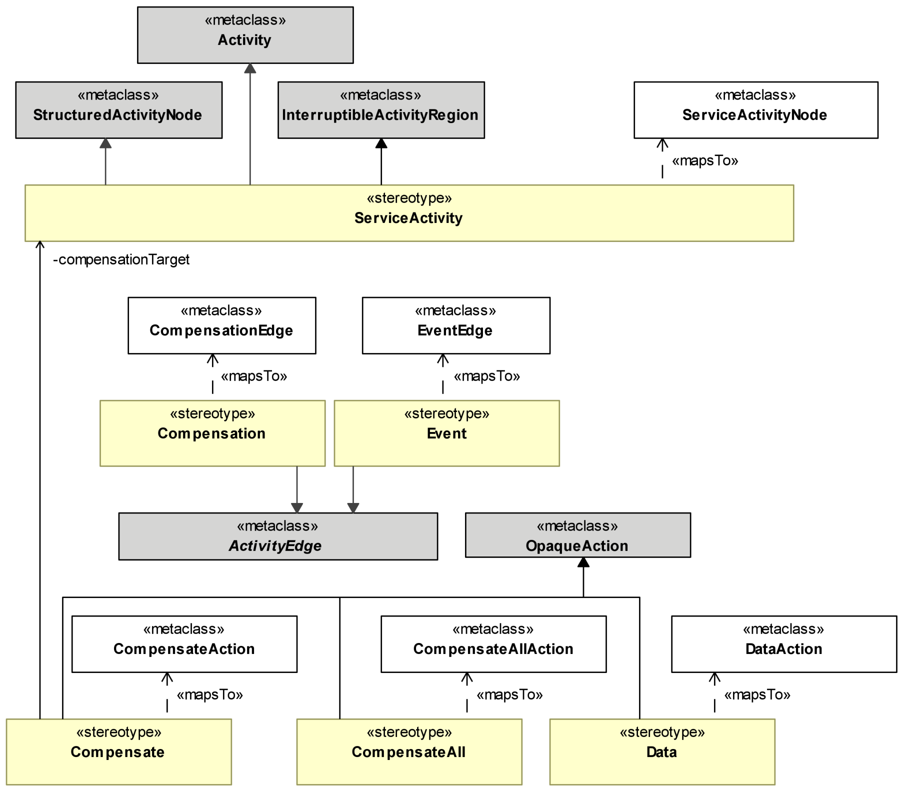

«metaclass» Activity
«metaclass» StructuredActivityNode
«metaclass» InterruptibleActivityRegion
«metaclass» ServiceActivityNode
«mapsTo»
«stereotype» ServiceActivity
-compensationTarget
«metaclass» CompensationEdge
«metaclass» EventEdge
«mapsTo»
«mapsTo»
«stereotype» Compensation
«stereotype» Event
«metaclass» ActivityEdge
«metaclass» OpaqueAction
«metaclass» CompensateAction
«metaclass» CompensateAllAction
«metaclass» DataAction
«mapsTo»
«mapsTo»
«mapsTo»
«stereotype» Compensate
«stereotype» CompensateAll
«stereotype» Data

Figure 3.10: UML4SOA Stereotypes for Structuring Classes

3.2.3.1 Structuring Classes

We begin with the structuring classes of the UML4SOA meta-model. Figure 3.10 shows the mapping of these classes of the UML4SOA meta-model to the stereotypes of the UML4SOA profile.

The most important stereotype is ≪*ServiceActivity*≫, which corresponds to the `ServiceActivityNode` meta-class. The ≪*Compensation*≫ and ≪*Event*≫ stereotypes are based on `ActivityEdge` and correspond to the `Compensation-Edge` and `EventEdge` meta-classes of UML4SOA. Note that we do not define tags here; the relationship between base elements and compensation/event handlers is given through the standard meta-attributes of `ActivityEdge`; constraints apply as per definition in the UML4SOA meta-classes `ServiceActivityNode` and `ActivityEdge`.

In the lower section of the figure, three stereotypes for actions are defined, namely ≪*Compensate*≫, ≪*CompensateAll*≫ and ≪*Data*≫. They correspond to the meta-classes `CompensateAction`, `CompensateAllAction` and `DataAc-`

`tion`, respectively. The first of these needs a tag definition: The ≪*Compensate*≫ stereotype must be tagged with the target service activity to be compensated.

3.2.3.2 Communication Classes

We continue with classes used for communication, i.e. the various sending and receiving actions as well as pins for specifying partners and data. Figure 3.11 shows the mapping of the corresponding meta-classes to stereotypes.

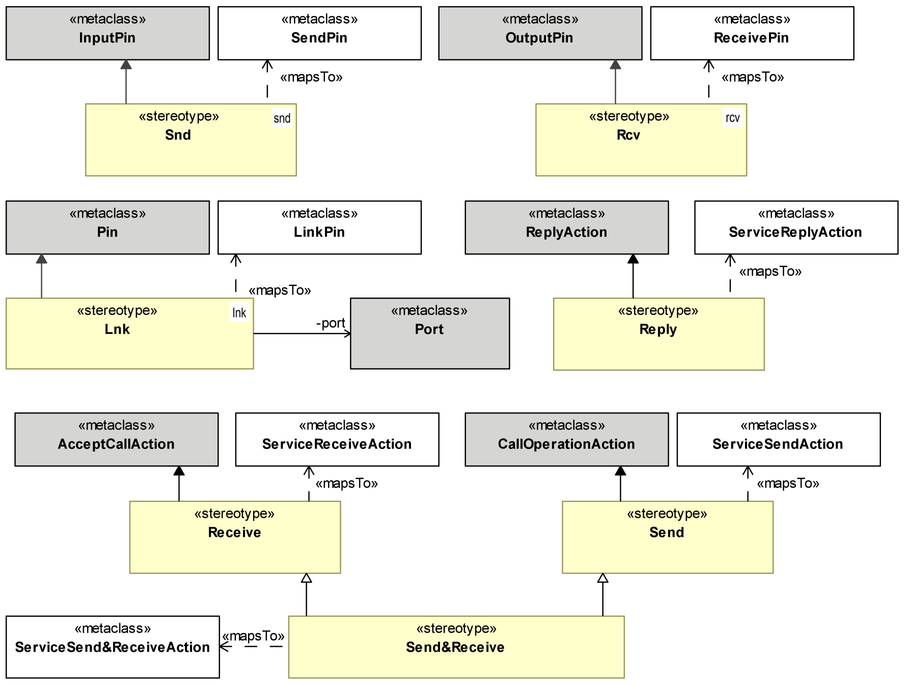


Figure 3.11: UML4SOA Stereotypes for Communication Classes

On the top, the stereotypes ≪*Snd*≫ and ≪*Rcv*≫ are defined; both with an optional icon for displaying a pin graphically. The first stereotype maps to the `SendPin` meta-class, thus inheriting the requirement that the type of the pin must be either a `MessageType` or an `PrimitiveType`; the second stereotype likewise maps to the `ReceivePin` meta-class.

In the second row, the stereotype ≪*Lnk*≫ is mapped to the meta-class `LinkPin`. Note that we need to define an additional tag here, specifying which port of the corresponding participant is referenced.

The ≪*Reply*≫ stereotype in the second row and the ≪*Receive*≫, ≪*Send*≫, and ≪*Send&Receive*≫ stereotypes in the third row are used for tagging communicating actions. They correspond to the `ServiceReplyAction`, `Service-`

`ReceiveAction`, `ServiceSendAction` and `ServiceSend&ReceiveAction` meta-classes, respectively. No tag definitions are required, however, once again, constraints apply.

3.2.3.3 Protocol Classes

Finally, we get to the last five stereotypes of the UML4SOA profile, which concern the specification of service protocols and are shown in figure 3.12.

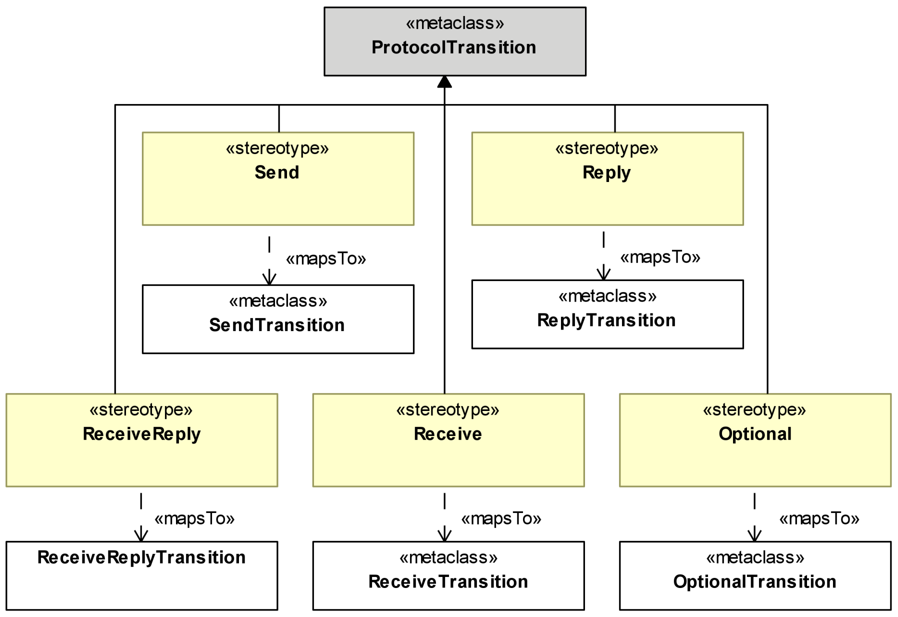


Figure 3.12: UML4SOA Stereotypes for Protocol Specification

We have seen three of the stereotypes listed here before; they are used for both activities and protocol state machines. As expected, the ≪*Send*≫ stereotype maps to the `SendTransition` meta-class, ≪*Receive*≫ maps to the `ReceiveTransition` meta-class, and ≪*Reply*≫ maps to the `ReplyTransition` meta-class. Furthermore, the ≪*ReceiveReply*≫ stereotype maps to the `Receive-ReplyTransition` meta-class and the ≪*Optional*≫ stereotype maps to the `OptionalTransition` meta-class.

No further tag definitions are required, though it is worth noting that the communicating stereotypes inherit a constraint on the trigger allowed on a stereotyped transition. By contrast, the ≪*Optional*≫ stereotype is only used for tagging the transition.

The UML4SOA profile is thus complete and can be used in arbitrary, profile-enabled UML modelling tools. An example for this is given in section 3.4.

3.2.4 Data Handling

An important point in modelling service behaviour and service orchestrations is data handling. Data is received by services, manipulated, and then sent on or back to another service. We have devised a declarative, textual language for this purpose, which aims to closely match the level of detail of UML4SOA. A major goal of the UML4SOA data handling language was to be generic enough to be understandable on the modelling level, yet contain enough information to allow transformation to more lower-level languages for execution. The main requirements for a UML4SOA data handling language is support for data in messages sent in-between services, and variables for storing such data, which requires:

- Support for primitive and complex (composite) data types.
- Typing of and access to variables, including assignments and partial assignments.
- Basic operations for manipulation data.

The UML4SOA data handling language is strongly typed and based on UML primitive types as well as classes annotated with the ≪*MessageType*≫ stereotype from the SoaML profile, which are in effect data types (i.e., classes without behaviour). Data is modified by imperative statements which may be used in three distinct areas within UML4SOA models:

- *Pins.* While receive pins may only hold variable references or (implicit) variable declarations, send pins may also be used to construct new data on-the-fly.
- *Guards.* A guard may contain a boolean-typed UML4SOA expression.
- *Data Handling Actions.* When inline data handling in pins or guards is not possible, data handling statements can also be added explicitly with a ≪*Data*≫ action.

Usually, services and service orchestrations directly work on structured data, i.e. ≪*MessageType*≫-typed classes which carry the business-relevant information. The UML4SOA data handling language provides built-in support for these data types, although some restrictions apply:

- A data type may only be assembled from primitive types or other structured data types.
- Inheritance is allowed, but again only amongst structured data types.
- Associations between structured data types is possible with the exception of bidirectional associations.

The data manipulation language supports both sets and lists (unordered and ordered associations). Furthermore, operations on basic data types is supported (mathematical operation on integers and reals; logical operations on booleans, and concatenation on strings).

3.2.4.1 Syntax Used

Whenever a concrete syntax is described in this document, we display it the same manner as in the Java Language Specification [GJSB05]. We use a context-free grammar, i.e. a number of productions with a nonterminal symbol on the left and both terminals and non-terminals on the right:

Listing 3.2: Context-Free Grammar Example

Element:
    execute(*AnotherElement***)**

We denote nonterminal symbols with *italics*, and terminal symbols with **bold** font. A definition of a nonterminal is given by the nonterminal suffixed with a colon (:), as shown for *Element* in the example. For *Element*, the right-hand side consists of the terminal symbol **execute** followed by the terminal symbol for an opening brace (**(**), the non-terminal *AnotherElement*, and another terminal symbol, the closing brace (**)**).

In general, the right-hand side of a non-terminal definition consists of one or more lines which form the possible alternatives. An example is the following definition:

Listing 3.3: Denoting Alternatives

Elements:
    Element
    Elements Element

This definition of *Elements* introduces two alternatives: Either the non-terminal *Element*, or *Elements* again, followed by a single *Element*. This definition is recursive, as *Elements* occurs both left and right of the colon. Note that if a line is too long to fit on the page, we let it continue indented below.

We introducing a special suffix (*opt*) for specifying an optional element. Consider the following example:

Listing 3.4: Optional Elements (1)

Element:
    execute(*AnotherElement***)** $OtherElements_{opt}$

In this definition of *Element*, *OtherElements* may optionally be specified after the main **execute** definition. This is a shortcut for

Listing 3.5: Optional Elements (2)

Element:
  execute(*AnotherElement***)**
  execute(*AnotherElement***)** *OtherElements*

Finally, we define three special non-terminals:

- *String*, which identifies a sequence of arbitrary characters,
- *Number*, which identifies a sequence of characters in the range of `[0-9]`,
- and *VarName*, which identifies a sequence of letters and numbers, where the first character must be a letter.

This concludes the introduction of the syntax notation.

3.2.4.2 Grammar

We start with the declaration of a statement and preliminaries:

Listing 3.6: UML4SOA Data Handling Syntax

DataHandling:
  Statement

Statement:
  Declaration
  Assignment

Declaration is used to declare the type of a variable. To denote an ordered or unordered list, the corresponding brackets ([] or {}) can be appended to the type:

Listing 3.7: UML4SOA Data Handling Syntax: Declarations

Declaration:
  *Type Identifier***;**

Type:
  VarName $([]_{opt} \mid \{\}_{opt})$

Identifier:
  VarName

An *Assignment* is an expression for assigning a value to a variable. It is split between a left-hand-side (left of the assignment operator) and a right-hand-side (right of the assignment operator):

Listing 3.8: UML4SOA Data Handling Syntax: Assignments

```
Assignment:
    LeftHandSideExpression := RightHandSideExpression;
```

Left-hand sides are, in effect, references to variables or elements within variable types. A variable in UML4SOA has a declaring scope, which is the service activity it was first used or declared in. If the scope of a variable is not the current one, it may be given with the **::***ServiceActivityName* syntax. Furthermore, not only a variable can be referenced but also a part within the variable, which must be a publicly accessible field of a ≪*MessageType*≫.

Listing 3.9: UML4SOA Data Handling Syntax: Left-Hand Sides

```
LeftHandSideExpression:
    (:: VarName.)opt VarAccess

VarAccess:
    VarName (. VarAccess)opt
```

Right-hand sides are more complex as they can be used not only in assignments, but also in conditional statements; furthermore, they contain the complete syntax for data manipulation and calculations. *RightHandSideExpression* is defined by starting with a conditional or, which has the least precedence, and continuing until we reach the basic literals and qualifiers.

Listing 3.10: UML4SOA Data Handling Syntax: Right-Hand Sides (1)

```
RightHandSideExpression:
    ConditionalOrOperation

ConditionalOr:
    ConditionalAnd (|| ConditionalAnd)opt

ConditionalAnd:
    Equality (&& Equality)opt

Equality:
    Relational ((== | !=) Relational)opt
```

Listing 3.11: UML4SOA Data Handling Syntax: Right-Hand Sides (2)

Relational:
    Additive $((> \mid >= \mid <= \mid <)$ *Additive*$)_{opt}$

Multiplicative:
    PrefixedUnary $((* \mid / \mid \%)$ *PrefixedUnary*$)_{opt}$

PrefixedUnary:
    $(- \mid !)$ *Unary*

Evaluating right-hand sides starts from the bottom to the top, i.e. the prefixed unary literals **-** and **!** have the highest precedence, while the conditional or || has the lowest.

Before we can define the literals, we have to take care of the unary elements referenced above. A unary element is either a literal, a left-hand-side expression, or a right-hand-side expression in parenthesis.

Listing 3.12: UML4SOA Data Handling Syntax: Unary Elements

Unary:
    Literal | *LeftHandSideExpression* | *ParenthesisExpression*

ParenthesisExpression:
    (*RightHandSideExpression***)**

Finally, we can define the literals, which are simple numbers, string constants, boolean constants, or the special value **null**.

Listing 3.13: UML4SOA Data Handling Syntax: Literals

Literal:
    StringLiteral | *NumberLiteral* | *BooleanLiteral* | **null**

StringLiteral:
    "*String*"

NumberLiteral:
    Number **(.** *Number*$)_{opt}$

BooleanLiteral:
    true | **false**

As usual, this grammar allows a few constructs which are not legitimate from a semantic point of view. For example, comparing a string with a boolean using the *Equality* construct makes no sense in a strongly typed language. We believe, however, that these cases are intuitively clear and thus do not require further lengthy discussion.

3.2.4.3 Using the data language

Three of the non-terminal elements of the UML4SOA data language can be used as *top-level* elements in expressions inside of UML4SOA models:

- The *DataHandling* statement is intended to be used in ≪*Data*≫ actions. A ≪*Data*≫ action may contain an arbitrary number of data handling statements.
- *LeftHandSideExpression*s and *Declaration*s may be used in ≪*Rcv*≫ pins. The first is used to specify an already existing variable in which the data received is to be placed. The second can be used as a shortcut for the modeller; it both declares the variable and specifies it for the received data.
- In ≪*Snd*≫-Pins, entire *RightHandSideExpression*s can be used. This allows creating data on-the-fly. In this case, it is convenient to think of a remote message invocation as a distributed assignment.

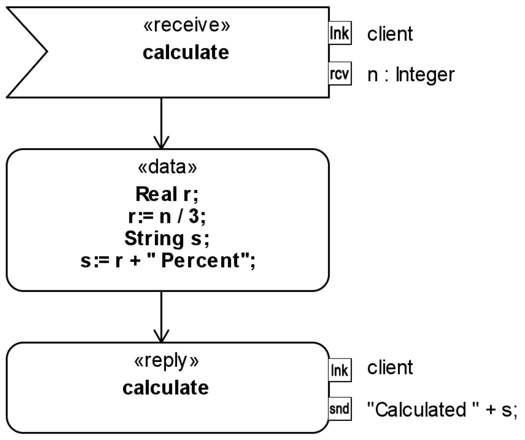


Figure 3.13: UML4SOA Data Manipulation: Simple Example

An example of using simple data types, assignment operations, and basic operation on numbers and strings is shown in figure 3.13. First, a variable is declared on-the-fly in a ≪*Rcv*≫ pin of the ≪*Receive*≫ action `calculate`; the variable is called `n` and is typed with the well-known UML type `Integer`. Second, a data handling action is used which executes four statements:

- The variable `r` is declared with the UML type `Real`,

- `r` is assigned the expression `n / 3`,
- the variable `s` is declared with the UML type `String`,
- `s` is assigned the expression `r + " Percent"`.

Finally, the reply action uses the on-the-fly right-hand expression `"Calculated " + s` to add an additional string before `r`, the result of which is then sent back to the invoker.

As indicated above, the UML4SOA data language also provides extensive support for dealing with structured data types, which are tagged with ≪*MessageType*≫ in SoaML. An example of three structured data types is shown in figure 3.14: a `Thesis` object may reference a `Student` and a `Tutor`.

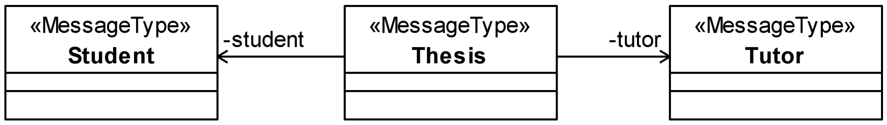


Figure 3.14: UML4SOA Data Manipulation: Structure Types

Working with the instances of the `Thesis` class in UML4SOA can take advantage of these associations. Figure 3.15 shows an example where both the tutor and the student association ends are set in a single data action.

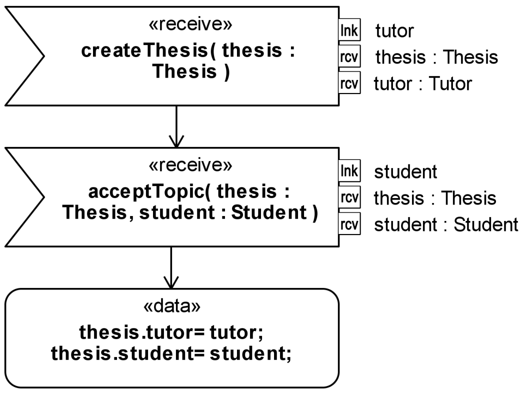


Figure 3.15: UML4SOA Data Manipulation: Structure Example

To sum up, adding a data handling language to UML4SOA has the benefits of being able to specify data operations on the UML level of abstraction. With its low complexity and easy-to-use syntax, the language is a good match for the UML4SOA graphical language and will later enable transformation to code (chapter 6).

3.2.5 Changes to the UML

Specifying service behaviour in UML introduces some key new requirements for a modelling language which was originally designed with object-oriented systems in mind. The SoaML profile [OMG09b] has already shown how to mould the UML to include SOA concepts, which has led to the proposal of adding the concept of *conjugation* to the UML.

Within UML4SOA, we have identified the need for two additional changes to the UML to enable developers to model SOA behaviour in a natural and straightforward way. In the following, we revisit these changes already introduced in the previous sections.

3.2.5.1 Adding an InputPin to AcceptCallAction

UML4SOA uses the UML meta-class `AcceptCallAction` with the stereotype ≪*Receive*≫ for denoting a place where service behaviour waits for an incoming call through a port of the corresponding participant. As has been discussed above, the concrete port must be specified as part of the action to indicate the partners from which a call is to be expected. UML4SOA has introduced the ≪*Lnk*≫ stereotype for this purpose, which is attached to the UML meta-class `InputPin`, as the partner information is an input to the receiving action.

Unfortunately, the UML superstructure [OMG10b] contains a restriction on `AcceptEventAction`, which is a superclass of `AcceptCallAction`, which prevents the use of input pins on instances of this class.

UML4SOA requires that this restriction is relaxed to allow pins which carry additional information for the receiving action, which, in our case, is a ≪*Lnk*≫-stereotyped `InputPin` specifying the port the operation attached to the trigger of the action is received on.

3.2.5.2 Allowing Call Observations in PrSMs

Transitions in UML Protocol State Machines [OMG10b] are based on the meta-class `ProtocolTransition`. This class contains two important restrictions. First, the `effect` association must be empty, i.e. a protocol transition may not have associated actions. Second, as a subclass of `Transition`, a protocol transition may include a trigger. There are two restrictions on this trigger:

- First, the specification of `ProtocolTransition` includes the requirement that if a call trigger is used, the operation referenced *should apply* to the context classifier of the state machine of the protocol transition.
- Second, the specification states that non-call events may be used on protocol transitions, but again refers to *incoming* events whose target is the context classifier.

We believe that in the context of service protocol specification, this restriction should be lifted to be able to observe events which *originate* from the context classifier instead of using it as a target. In fact, a corresponding UML meta-class for this concept exists: `SendOperationEvent` specifies that a call invocation request is sent to an object (at which it may result in the occurrence of a call event).

As sending out calls to partner services requires being able to note this fact in a protocol, we believe that it should be possible to also reference operations which are *used* by the context classifier of a protocol state machine. UML4SOA thus extends the ability of PrSMs to include triggers with a `SendOperationEvent` event. It is important to note that this event is not an *effect* of a transition; rather; it is an *observed operation call* of the participant the classifier of the PrSM is attached to.

This ability is restricted in UML4SOA to transitions stereotyped with ≪*Send*≫ or ≪*Reply*≫.

3.2.6 UML4SOA/Open and UML4SOA/Strict

In this section, we introduce two *dialects* of UML4SOA: One serves modellers interested in having maximum freedom in applying UML4SOA in combination with the UML, while the other serves modellers interested in code generation and formal analysis.

On the one hand, UML as a graphical language is great for communication between people. For this use case, the focus lies on readable diagrams, which tend to focus on the overall architecture of a system and ignoring low-level details. Some of the diagram types of UML, for example use case diagrams, are explicitly geared towards this usage, but with a sufficient level of abstraction this method is applicable to all modelling elements. UML4SOA can be used for this purpose: UML4SOA/Open defines a dialect which contains no restrictions on how UML elements and UML4SOA elements may be used and combined in models, only requiring the constraints in section 3.2 to hold.

On the other hand, model-driven software approaches build on generating code from models. To enable such code generation, the semantics of the models must be specified more precisely, which in general requires more detail and stricter rules for placing elements in the model. Again, some diagram types in UML are better suited for this purpose, for example state machines and activity diagrams, but once more there are also methods for generating i.e. tests from use case diagrams. UML4SOA can be used for this purpose as well: UML4SOA/Strict defines a set of rules for modellers to follow which enables formal analysis and code generation which will be detailed in chapters 5 and 6.

3.2.6.1 UML4SOA/Open

The purpose of UML4SOA/Open is to give maximum freedom to software modellers. For this reason, no additional constraints apply — UML4SOA elements may be freely mixed with UML activity and state machine model elements, fully exploiting the means of specifying models with UML.

3.2.6.2 UML4SOA/Strict

By contrast, UML4SOA/Strict defines a set of rules which must be followed to create compliant UML4SOA activity and state machine models usable for generation of code and the specification of a formal semantics.

- A UML4SOA/Strict model must be based on a SoaML ≪*Participant*≫ with ≪*Service*≫ or ≪*Request*≫ ports. Each port must have a port type stereotyped with ≪*ServiceInterface*≫ which may include operations (either directly or inherited) and declare usage relationships to other types. Each operation may have multiple `in` and `return` parameters; `out` and `inout` parameters are not allowed.

- All UML4SOA activities must be stereotyped with ≪*ServiceActivity*≫ and attached to a ≪*Participant*≫. The only actions allowed in an UML4SOA activity diagram are the actions stereotyped with UML4SOA stereotypes with the one exception of `RaiseExceptionAction`. All communicating actions must reference an operation (either directly or through a trigger) from one of the port types of the corresponding participant.

- For controlling the workflow, decision and merge nodes as well as fork and join nodes may be used. However, the resulting model must be well-nested, i.e. all paths from a decision node not ending in a flow-final or activity-final node must end in a merge node. The same goes for fork and join nodes. Loops can (as usual) be modelled using fork and join nodes; however, the looping (back) link may not carry any additional actions. Each service activity must have an identifiable start node, i.e. either an action without incoming links or a dedicated start pseudo node.

- The only grouping constructs allowed are UML4SOA service activities. Handlers (again, service activities) may be attached as usual to service activities, but interrupting edges are restricted to non-handler service activities. Handlers may only be attached to non-handler service activities.

- Each event handler must start with a ≪*Receive*≫ action and end with a ≪*Reply*≫ action or a `RaiseExceptionAction` to ensure the event handler termination is either communicated to the partner, or exception handling is triggered.

- All data handling statements in ≪*Snd*≫ and ≪*Rcv*≫ pins, guards, and data actions must follow the syntax and semantics of the UML4SOA data manipulation language. The only UML primitive types allowed are `Integer`, `Boolean`, and `String`; additionally, floating point values may be declared using a (custom) data type `Double`, and dates with a (custom) data type `Date`.

- The root behaviour of a participant must start with a ≪*Receive*≫ action to ensure that it can be started from the outside.

For protocol state machines, the following rules apply:

- Transitions not annotated with a UML4SOA communication stereotype are allowed, but are assumed to be internal to the protocol and the corresponding implementation. They do not follow the usual completion semantics.
- States may not be nested, i.e. the state and transition structure must be flat.
- An explicit start pseudo node is required to be present. The usual restrictions apply for the start transition.

We believe that besides being a requirement for code generation and formal analysis, these requirements also lead to more readable diagrams and easier implementation, and thus recommend following the constraints given here regardless of the use of code generation.

3.2.7 Lifecycle Management

A SoaML participant with its accompanying service and request ports and UML-4SOA activities models a single instance of an orchestration execution. The owned behaviours (UML4SOA service activities) each model one execution of the orchestration; the port protocols (UML4SOA PrSMs) each model an observation of the interaction with a partner during the lifetime of the orchestration (provided or requested).

However, in client/server and SOA computing, an orchestration is normally executed multiple times, often in parallel. In UML4SOA, we do not require the developer to model this bootstrapping mechanisms as it is normally not of interest to the modeller. Handling of multiple workflows, including instance matching, is done implicitly in UML4SOA designs as follows (cf. figure 3.16).

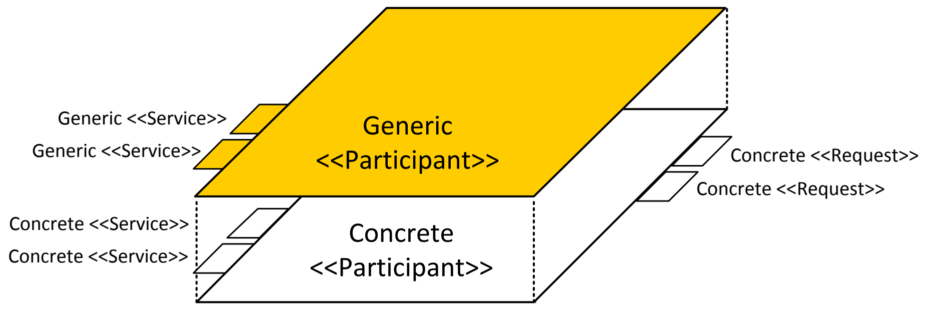


Figure 3.16: Generic and Concrete Participants

A concrete UML4SOA participant (i.e. the version modelled by a user) is not instantiated directly on system startup. Instead, one can imagine that a *generic version* of the participant is implicitly created and instantiated. The generic version has as many ≪*Service*≫ ports as the concrete one — these are *generic* ≪*Service*≫ ports, which each provide and accept only one operation again and again, which corresponds to the first accepted operation in the concrete ≪*Service*≫ port PrSM. On startup of the generic participant, the generic ports wait for incoming calls. Once a message is received, a new instance of the concrete participant along with its ports is instantiated by the generic participant and the startup message is passed to the corresponding *concrete* port.

All non-startup communication is done directly with the *concrete* instances of the participant and its ports. To ensure instance matching, the port instances are provided to the workflow via the ≪*Lnk*≫ pins.

Finally, instance matching, i.e. routing messages to the appropriate port and thus instances of UML4SOA activities, requires some information (like an ID) as part of the message with which the system can route a message to the appropriate port (and thus workflow) instance. This information is assumed to be added transparently to the incoming and outgoing messages of the workflow based on the unique IDs associated with each port instance.

Each service action in UML4SOA service activities has a ≪*Lnk*≫ pin which carries the information about the instance of the port which is in use for this particular orchestration instance. This information is used to match calls to and from the correct workflow instance.

3.3 Modelling Examples

In this section, we will detail how the thesis management scenario from the eUniversity case study from the Sensoria project has been modelled with UML-4SOA, and give some pointers to other examples.

3.3.1 Modelling the eUniversity Case Study

In section 3.1, we have introduced the static (SoaML) model of the eUniversity case study (Figure 2.9 on page 35). Now, after having discussed the UML4-SOA profile and its extensions for activities and protocol state machines, we can add the behaviour of the `ThesisManagement` participant and the protocols of its ports. In the following, we use the UML4SOA/Strict dialect for both the activity and the PrSM models.

The UML4SOA activity describing the behaviour of the `ThesisManagement` participant is shown in figure 3.17. From top to bottom, the behaviour is as follows:

- The orchestration begins with a ≪*Receive*≫ action, requiring a client to send the `createThesis` call via the service port `tutor`. A thesis object is expected and placed in the newly declared `thesis` variable.

- Afterwards, the process starts with its `Main` activity and another receive operation: `acceptTopic` allows a student to start working on the thesis. Attached to `Main` is an exception handler which catches the `Thesis-`

`FailedException`. It contains one action, namely a ≪*Compensate*≫ call with the target activity `Registration`.

- Having completed `acceptTopic`, the `Registration` activity is entered. Here, we first inform the examination office about the newly started thesis, and secondly register the student for a seat in the graduation gala. This is a classic example of how certain parts of a workflow complete successfully but may need to be undone later on; therefore, the `Registration` scope has an attached compensation handler `CompensateRegistration` which undoes the reservation of a seat in the gala (this happens if the thesis is not accepted).

- Finally, the tutor is notified that a student has accepted the thesis and is now working on it.

- Now that the thesis is in progress, we enter the `InProgress` activity, which starts with a loop. In this loop, the student is allowed to send updates by using the `updateStatus` call via the `student` port until the thesis is complete, in which case he sends a `finished` call.

- During this time, which is potentially quite long, the tutor might want to request updates. Therefore, the `InProgress` activity has an event handler, `StatusInformation`, which contains the receive action `getStatus` to be used by the tutor for retrieving the current status of the thesis.

- Once the `finished` call has been received, the assessment is requested from the tutor using `getAssessment`, which is reported to the student. In case the thesis was finished successfully, this information is reported to the examination office and the process ends. If not, a failure is reported and an exception thrown, which leads, via the compensate activity in the `ExceptionHandler`, to unregistering the student from the gala.

As the `ThesisManagement` participant uses four ports — two service port and two request ports — we define four UML4SOA protocol state machines for specifying the externally visible protocol of the participant.

Figure 3.18 shows the protocols of the `ThesisManagement` participant. From top left to bottom right, these are the `student` protocol, the `eoffice` protocol, the `bboard` protocol, and the `tutor` protocol.

Student Protocol

The protocol provided to the student allows receipt of the `acceptTopic` call. Once received, the protocol is in `working` state, allowing the receipt of either the `updateStatus` call, which leads back to `working`, and `finished`, which requests the process to finished (being replied to with ≪*Reply*≫ `finished`).

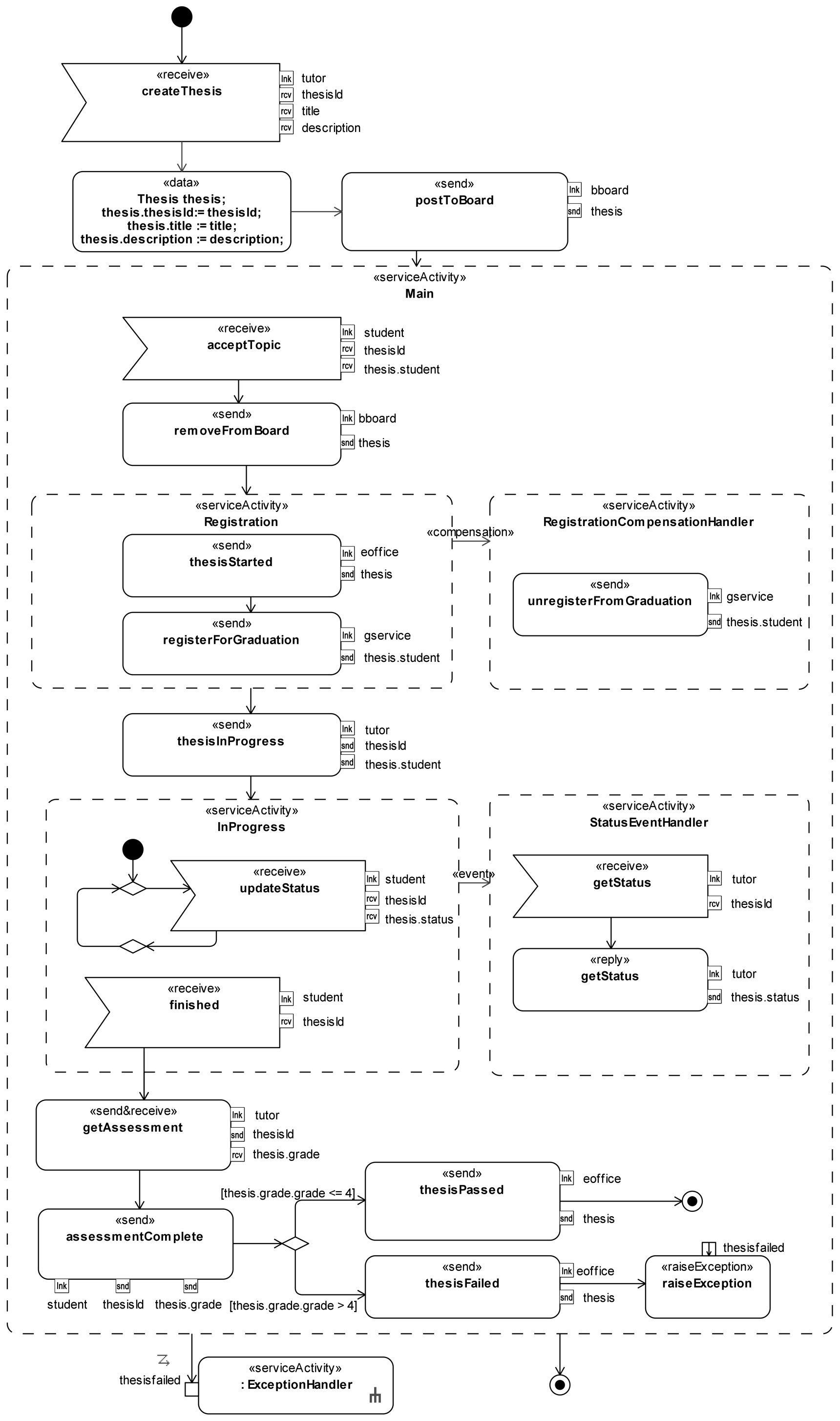


Figure 3.17: eUniversity Case Study: Thesis Manager Activity

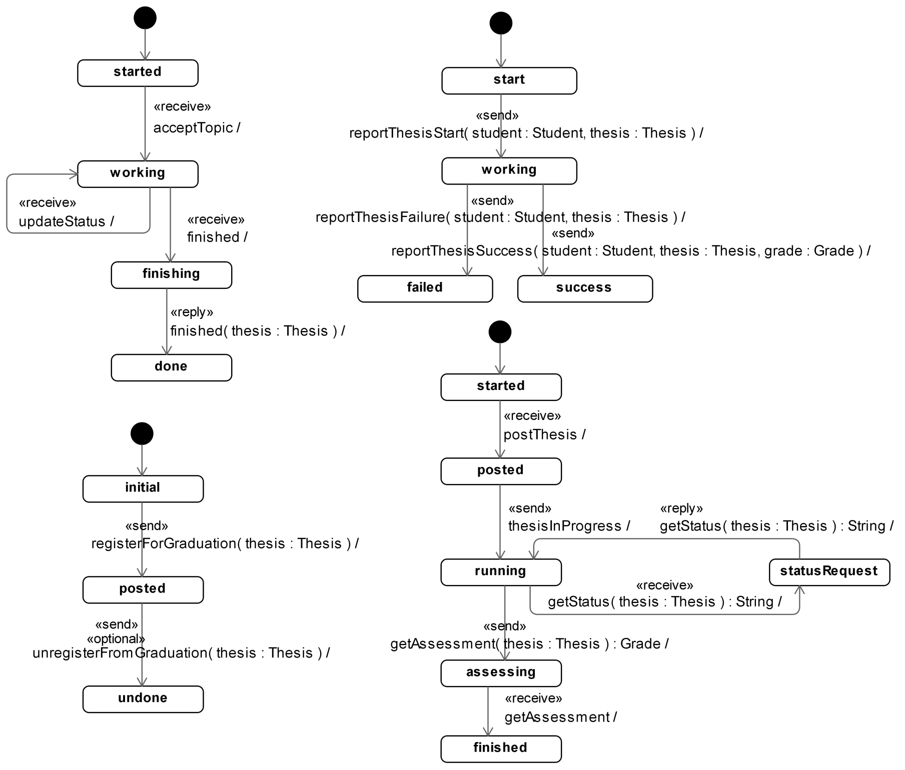

started
«receive»
acceptTopic /
working
«receive»
updateStatus /
«receive»
finished /
finishing
«reply»
finished(thesis : Thesis) /
done
initial
«send»
registerForGraduation(thesis : Thesis) /
posted
«send»
«optional»
unregisterFromGraduation(thesis : Thesis) /
undone
start
«send»
reportThesisStart(student : Student, thesis : Thesis) /
working
«send»
reportThesisFailure(student : Student, thesis : Thesis) /
«send»
reportThesisSuccess(student : Student, thesis : Thesis, grade : Grade) /
failed
success
started
«receive»
postThesis /
posted
«send»
thesisInProgress /
«reply»
getStatus(thesis : Thesis) : String /
running
statusRequest
«receive»
getStatus(thesis : Thesis) : String /
«send»
getAssessment(thesis : Thesis) : Grade /
assessing
«receive»
getAssessment /
finished

Figure 3.18: eUniversity Case Study: Protocol Specification

EOffice Protocol

The protocol of the examination office begins with the ability to send a `reportThesisStarted` call, indicating that a student has started a thesis. In the subsequent state `working`, the thesis is either reported as being successfully completed with `reportThesisSuccess` leading to the `success` state, or having failed, in which case we receive a `reportThesisFailure` which leads to the `failed` state.

Graduation Protocol

The graduation service protocol is rather simple — the `TutorManagement` participant expects to be able to send a `registerForGraduation` call to the graduation service, and it might — using the ≪*Optional*≫ stereotype — also need to unregister the student with `unregisterFromGraduation`.

Tutor Protocol

Finally, the tutor protocol is another protocol provided by the `ThesisManager`. The participant first expects a `postThesis` message from the tutor. Once a student has chosen the thesis, the tutor is informed with the `thesisInProgress` call. Now, the tutor may send `getStatus` calls for retrieving the status of the thesis, for which he must be able to receive a reply. Finally, once the `getAssessment` call is sent to the tutor, he must send back the assessment with the `getAssessment` call.

3.3.2 Other Examples

Besides the thesis management scenario of the eUniversity case study, UML4-SOA has also been used for several other scenarios from different case studies within the SENSORIA project.

- In the context of the eUniversity case study, another scenario has been modelled with UML4SOA, which revolves around a *student application* to an online university.

- From the automotive case from the SENSORIA project, several scenarios have been modelled in UML4SOA, the most elaborate of which is the *roadside assistance* scenario.

- Finally, the finance case study and its *credit request* scenario have been modelled with UML4SOA.

An overview of these scenarios is given in [EGK+10]. Furthermore, the SENSORIA web site `www.sensoria-ist.eu` contains tutorials and downloads for each of these case studies.

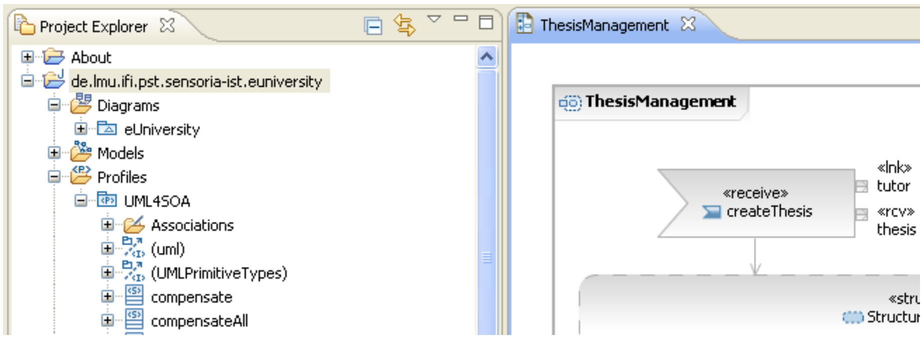


Figure 3.19: UML4SOA Tool Support in RSA

3.4 Tool Support

As outlined in chapter 2, one of the benefits of defining a UML profile instead of a heavyweight extension or a completely new graphical language is the availability of UML tools which readily offer support for UML profiles. The definition of profiles using the mechanisms provided by the tool is usually sufficient to be able to use the included stereotypes, although not always in the most convenient way.

In the next section, we discuss support for UML4SOA in IBM Rational Software Architect (RSA) [IBM09] and the MDT UML2 Tools [Ecl10d] by means of a simple profile definition. In section 3.4.2, we discuss a more thorough integration into the tool MagicDraw [NoM10].

3.4.1 Profile Support in RSA and MDT UML2

Both the Rational Software Architect by IBM and the MDT UML2 Tools are based on the Eclipse platform and a common meta-model of the UML modelled in the Eclipse Modelling Framework (EMF). Thus, although the graphical front-ends of these two tools are very different, the underlying model is (mostly) equivalent. In particular, profiles can be defined which can be used by both platforms.

An example of using RSA with the UML4SOA profile is shown in Figure 3.19; the view in Eclipse is similar. On the left hand side, the stereotypes of the profile are shown, which can be applied to a certain element in the diagram by drag and drop or context menu selection. On the right-hand side, the first two actions from the `ThesisManagement` can be seen in the diagram view.

3.4.2 Integration into MagicDraw

In contrast to both RSA and the MDT UML2 Tools, the MagicDraw product by NoMagic is not built on Eclipse nor the EMF UML2 meta-model discussed

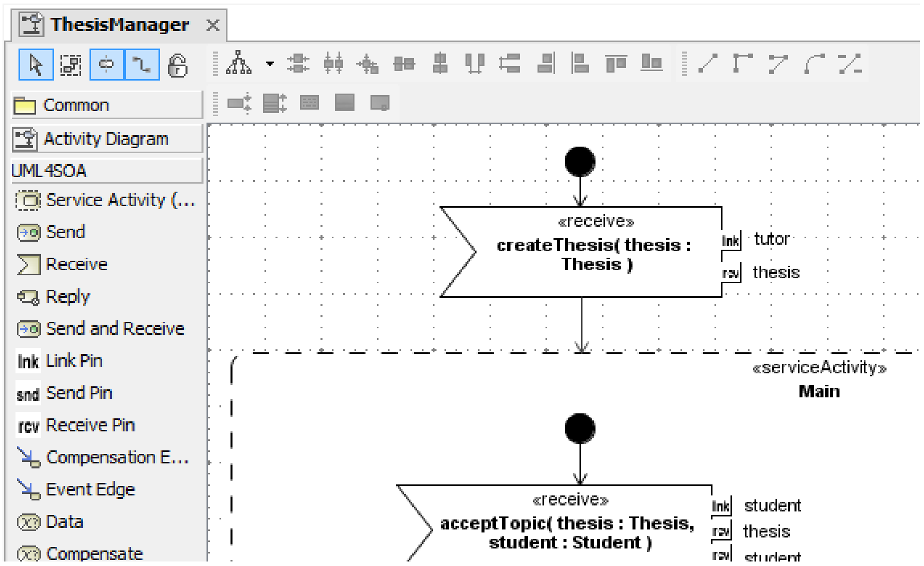


Figure 3.20: UML4SOA Tool Support in MagicDraw

above, but uses a proprietary UML meta-model. Thus, the profile must be defined separately using the tools provided in MagicDraw.

MagicDraw can be easily extended with additional *toolbars* which enable modellers to immediately create a stereotyped element without first having to select the correct meta-class and later on applying a stereotype. We provide such a toolbar for UML4SOA, which is shown in figure 3.20 on the left-hand side. The already stereotyped actions or edges shown there can be directly dragged to the diagram view on the right-hand side, which (again) shows the first two actions of the `ThesisManagement` orchestration.

3.5 Related Work

As the UML is the de facto standard for modelling software systems and furthermore offers the *profile* mechanism for allowing extensions, it is not surprising that a number of profiles and other extensions are available for modelling SOA systems. However, adding SOA semantics to the UML is not trivial: As shown in section 3.2.1, the UML semantics differs in several aspects from the ideas of SOAs. Section 3.5.1 discusses existing approaches which touch the area of behavioural SOA modelling.

UML models are not only about behaviour; and neither are profiles. The Sensoria project thus provides supplementary profiles which address additional aspects of SOA systems and can be used in combination with UML4SOA. They are discussed in section 3.5.2.

Although the aim of this thesis has been the adaptation of UML to behavioural SOA modelling, there are other languages which may be used for this purpose as well. An overview is given in 3.5.3.

3.5.1 UML extensions

[Joh05] provides an extension for the specification of services addressing structural aspects. The focus lies on the concepts of services and in particular technical concerns such as message passing with attachments. As such, it is closely related to the Web Service family standards; in fact, WSDL is mentioned explicitly. The profile thus does not include its own platform-independent layer. Furthermore, neither behaviour of services nor orchestration is addressed in that work.

The work of Skogan et al. [SGS04] has a similar focus to the approach presented in this thesis, i.e. a model-driven approach for services based on UML models and transformations to executable descriptions of services. The main difficulty in the use of this approach lies in the Web Service composition method, which is closely based on Web Services, as the workflow starts with a WSDL specification which is transformed to UML as the basis of a behavioural workflow. Furthermore, the stereotypes of the profile carry direct links to Web Service artefacts (for example, the ≪*WebServiceCall*≫ stereotype carries the corresponding WSDL definition in the tagged value `wsdl`). Thus, the relationship between Web Services and UML is much closer than in UML4SOA.

The style-based modelling and refinement approach [BHTV06, HLT03] by Baresi et al. focuses on modelling SOA architectures by refining business-oriented architectures. The refinement is based on conceptual models of the platforms involved as architectural styles, formalised by graph transformation systems. The extension includes stereotypes for the structural specification of services; however, it does not introduce specific model elements for behavioural descriptions of services.

Ermagan and Krüger [EK07] extend the UML2 with components for modelling services defining a UML2 profile for rich services. Collaboration and interaction diagrams are used for modelling the behaviour of such components. Neither compensation nor exception handling is explicitly treated in this approach.

Another approach to modelling of Web Service compositions is UML-S [DNsmGW08]. Although the authors explicitly mention Web Services instead of SOAs, the UML-S profile is defined in a relatively platform-independent way. The profile includes both static aspects and behavioural aspects. In the latter, however, the focus lies on providing stereotypes for control flow patterns such as `while` and `N-Join`. Very little information is given on the communicating actions, and neither event handling nor compensation handling is mentioned.

In 2006, the OMG started an effort to standardise a UML Profile and Metamodel for Services (UPMS) [OMG06]. SoaML [OMG09b] has been created in response to this call, and is on its way to becoming an OMG standard (the project is currently in its second beta version). Both UPMS and SoaML are restricted to the static aspects of SOA architectures. As such, SoaML has been used as the basis for the work in this thesis as already described in section 2.3.

Several approaches have been implemented for the automated transformation from UML to BPEL with the commonality of requiring very technical, BPEL-focused UML diagrams from designers. A first automated mapping of UML models to BPEL [AGGI03] defines a very detailed UML profile that introduces stereotypes for almost all BPEL activities — even for those already supported in plain UML, which makes the diagrams drawn with this profile hard to read. This profile is rather old; it is based on BPEL 1.1. and UML 1.4. A follow-up diploma thesis [Amb05] lifts it to BPEL 2.0 and UML 2.0, keeping the same spirit.

Another example which is directly based on BPEL (1.1) is the UML profile described in [Man03]. A state chart is used to denote behaviour, with additional nesting elements describing the partner who is responsible for an action. Advanced BPEL concepts such as event handling or compensation are not mentioned.

Finally, a rather new approach by Li et al. [LZP09] uses UML sequence diagrams to describe service compositions. Starting from a set of WSDL files, the sequence diagrams are annotated with BPEL-like stereotypes and are later transformed to BPEL. However, no profile has been defined, and neither compensation, event handling, nor faults are mentioned.

In contrast to the above approaches, UML4SOA does not focus on any particular target implementation technology. Rather, it attempts to provide, in the simplest and most UML-like way possible, a complete solution for modelling behavioural service-oriented concepts in UML.

3.5.2 Related Profiles

As sketched in the introduction of this thesis, UML4SOA has been defined in the context of the SENSORIA project [WBC$^+$09], which as a whole had the aim of supporting the rigorous engineering of service-oriented software. To be able to specify aspects of SOAs not covered in SoaML and UML4SOA, the project provides additional profiles [FGK$^+$10b] which may be used in combination with UML4SOA. These profiles address non-functional properties of services, business policies, implementation of service modes, and service deployment [FGK$^+$10a].

The non-functional extension profile for SOAs aims at modelling of arbitrary *quality of service* properties [GGK$^+$10]. The profile defines a general framework for quality of service (QoS), which can later be instantiated to address different concerns such as performance, logging, or security. In the same spirit as in UML4SOA, the NFP profile includes model transformations which support the automated code generation of middleware configurations with QoS constraints [GV10].

UML-based modelling of business policies in SENSORIA is based on the StPowla approach [GMRMS09]. StPowla defines a business process through basic tasks, which in turn use resources. Through access control over these resources, StPowla enables business stakeholders to adapt the process to arising needs based on policies. The UML extension for StPowla follows a hybrid approach: The actual policies are expressed using tables (outside of the UML), while the StPowla UML integration is a *support* profile integrating UML artefacts and policies.

Another approach developed within the SENSORIA project is the concept of *service modes* [HKMU06], which are an extension of *software architecture modes*. Service modes describe different architectural configurations of a SOA-based software system; depending on certain constraints, the system can dynamically switch between different modes to *adapt* to the current external requirements placed on its behaviour. Different modes may require different services based on different constraints. The UML profile for service modes [FUMK08] allows the specification of modes, collaborations and constraints as well as broker constraints and components to address this concern.

Finally, the *service deployment profile* [FEK^{+}07] of SENSORIA complements UML4SOA by providing the ability to specify service deployment on infrastructure nodes such as web servers and servlets. The profile allows the specification of characteristics a host server must provide, thus enabling verification of actual deployment configurations for safety and correctness.

3.5.3 Other modelling approaches

Behavioural service modelling can be performed on either a platform-independent or a platform-specific level; depending on the definition of *modelling*, one can even include traditional programming languages in this discussion.

A related approach to UML4SOA is the Business Process Modelling Notation (BPMN) [OMG09a]. BPMN is based on a workflow-like notation similar to activity diagrams. The BPMN language originates from the domain of business process modelling instead of software design specifications; the primary focus is thus a standard notation which is understandable by business stakeholders such as business analysts and managers, aiding them in their communication with one another and with the software engineers implementing the business workflows.

The BPMN specification includes an informal, partial mapping to BPEL. This mapping has shown to be problematic [RM06] due to conceptual mismatches between the two languages, which mainly revolve around the different backgrounds of business versus software engineering perspectives.

UML4SOA differs from BPMN in the same spirit. Firstly, its main target users are software developers. Second, although UML4SOA can be used to model business processes, we provide no specific support for them; instead, the modelling approach is to provide a generic, platform-spanning approach to modelling behaviour on a service level of abstraction.

Another major industry approach to SOA is the Service Component Architecture (SCA) [MR09]. SCA is a *programming model* for building applications based on SOAs. Different than in the case of Web Services, SCA does not focus on any particular language or framework; in fact, existing languages can be used to define the behaviour of services in SCA. The current standard contains specifications for Java, C++, BPEL, and PHP. As such, SCA does not include its own means of specifying service behaviour.

There are also *platform-dependent* approaches to the specification of service behaviour. The first language which comes to mind is the Business Process Execution Language (BPEL) [OAS07], which is specific to the domain of Web Services as it is fundamentally based on WSDL. BPEL itself is not a graphical language; however, most BPEL modellers introduce proprietary graphical modelling elements to aid customers in the specification of BPEL processes, whose syntax is in fact defined in XML.

Another platform-specific language for service modelling is the jBPM Process Definition Language (JPDL) [JBo10]. jBPM is a business process management suite based on process descriptions which can directly be executed by a virtual machine. JPDL is a process language which allows the specification of business processes using the Java language; similar to activity diagrams, it takes a graph-based approach to the specification of control flows. JPDL programs are deployed on the JBoss jBMN server to execute.

UML4SOA differs from the above approaches by providing a platform-independent, graphical modelling approach based on the Unified Modeling Language. Regarding SCA, UML4SOA models may be in fact be used with a number of target languages including Java and BPEL to contribute components to systems based on the SCA.

3.6 Summary

This chapter has introduced the UML4SOA profile, a lightweight extension of the UML for modelling the behaviour and the protocols provided and required of participants in service-oriented architectures.

In section 3.1, we have discussed the need for an extension for service behavioural modelling in the UML due to insufficient representation of key service concepts such as communicating actions, long-running transactions, and self-descriptions in activities and protocol state machines.

Section 3.2 has then introduced the UML4SOA meta-model and, subsequently, the profile. According to the main aim of the definition of the UML4SOA profile — minimalism and conciseness — we have defined additions to the UML for both activities and PrSMs, while reusing existing UML constructs such as structured activities, actions, and control structures such as fork or decision nodes.

This section has also discussed a lightweight data manipulation language for guards, actions, and pins in UML4SOA, which enables the modeller to stay on the same level of abstraction as in the rest of UML4SOA. Finally, in order to accommodate different usage scenarios of UML4SOA, the two *dialects* UML4SOA/Open and UML4SOA/Strict have been introduced. The former focuses on maximum expressiveness and integration with existing UML constructs, while the latter adds a set of constraints for ensuring unambiguous models ready for code generation and analysis.

Section 3.3 has shown a practical example of how to model with UML4SOA in the form of diagrams for the eUniversity case study introduced in chapter 1. More examples can be found in [EGK$^+$10].

We have discussed tool support for modelling UML4SOA in different UML modelling tools in section 3.4. Finally, section 3.5 has discussed related work.

Chapter 4

The Service Meta-Model

The MDD4SOA approach to model-driven development of service-oriented architectures consists of three main components — modelling, analysis, and code generation. Before discussing analysis and code generation in the next two chapters, we take a step back to address the underlying structure of these three components: A meta-model for services.

The previous chapter has already introduced UML4SOA, a profile for specifying SOA behaviour based on the UML. Together with parts of the UML and the SoaML profile referenced in chapter 2, UML4SOA defines a syntax for modelling SOA systems. This chapter refines this field with the introduction of the Service Meta-Model (SMM). The SMM is a generic, language- and platform-independent meta-model for describing the static, dynamic, and data handling aspects of a SOA system. It relates to the three main components of MDD4SOA as follows:

- Taken together with the restrictions discussed in UML4SOA/Strict, the UML, SoaML, and UML4SOA form the concrete syntax for the SMM.
- A semantics and corresponding formal analysis methods for the behavioural parts of the SMM allow evaluation of UML4SOA models, which will be discussed in chapter 5.
- Model transformations based on the SMM enable code generation of UML-4SOA models; in particular, we discuss transformations to actual executable code in Java and BPEL (chapter 6).

This chapter is structured as follows. First, we show how the individual parts presented in this thesis fit together based on the SMM (section 4.1). Second, we introduce the Service Meta-Model (SMM), detailing its design ideas (section 4.2). Finally, we conclude in section 4.3.

Published results: Results presented in this chapter are based on publications [MSK08a] and [MSK08b].

4.1 Overview

The three main contributions areas of MDD4SOA — modelling, analysis, and code generation — share a common idea of a service-oriented system. This chapter defines this common idea with the Service Meta-Model (SMM).

Before we introduce the SMM, it is important to understand how the SMM relates to UML4SOA, the formal semantics and analyses, and the transformations to code. This is shown in figure 4.1.

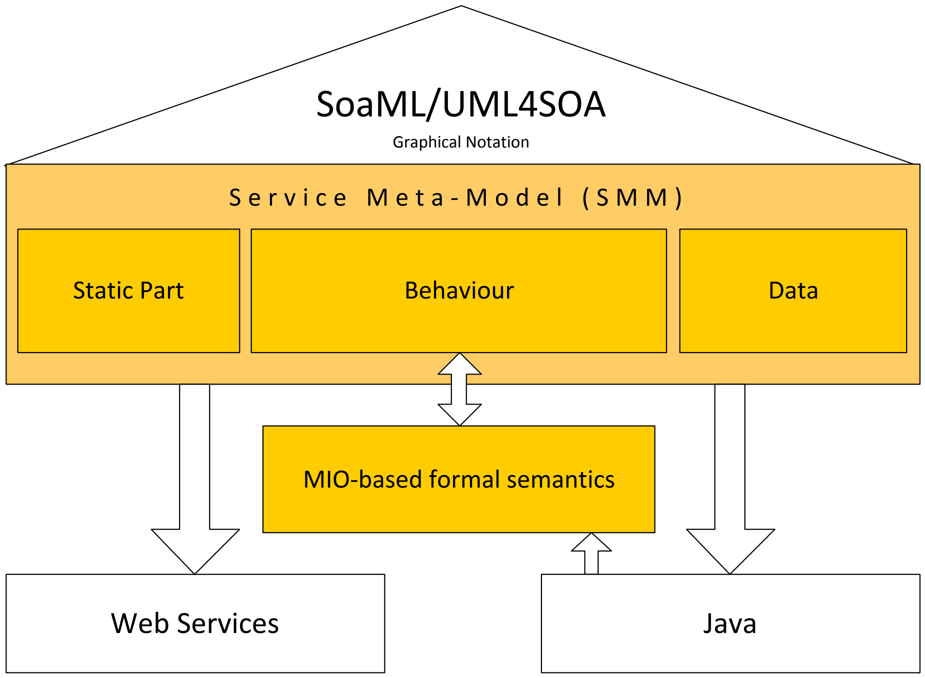


Figure 4.1: Overview: The SMM and Related Artefacts

The figure shows the Service Meta-Model (SMM) in the middle, consisting of three parts — a static part (left) which describes the static structure of a SOA system, a data part (right) which contains constructs for assignments, variable declarations, and operations, and finally (middle) the behavioural part.

The SMM in general does not have any notation, graphical or otherwise. This part is played by the UML and the SoaML and UML4SOA profiles, which are displayed on top of the SMM. The SoaML profile has been introduced in chapter 2 and is used as the concrete syntax for modelling the static part of the SMM. The UML4SOA profile has been introduced in chapter 3 and is used as the concrete syntax for the behavioural and data parts of the SMM.

We have introduced two variants of UML4SOA: UML4SOA/Open, which is intended for maximum expressiveness and free modelling of SOA systems mainly for communicating ideas, and UML4SOA/Strict, which requires a more rigid structure and constrains the elements allowed in the models. The SMM forms the basis for UML4SOA/Strict; strict-compliant models can thus be directly parsed into instances of the SMM. A complete description of this translation will be presented in chaper 6.

The behavioural part of the SMM is given a rigorous formal semantics in the domain of modal input/output automata. By means of a denotational semantics function, this translation is described fully in chapter 5. To simplify this specification, the SMM includes a textual notation for just the behavioural part, which will be introduced later in this chapter.

Finally, the SMM serves as the basis for the transformations to the Web Services family of standards (BPEL, WSDL, XSD) and Java, which are displayed at the bottom of the figure. In case of the Java implementation, a simulation and annotation approach allows executing the generated code and comparing runtime traces with the formal model, which is shown with an upward arrow. This approach is described in chapter 7.

In the following section, we give an introduction into the SMM.

4.2 The Service Meta-Model

The Service Meta-Model (SMM) is a generic meta-model for modelling SOA participants with provided and required services, service behaviour, and message exchanges between services.

The SMM attempts to capture general concepts which underlie service-oriented architectures while keeping in mind the requirements for concrete notations such as SoaML and UML4SOA as well as the need to be able to specify a rigorous formal semantics for the behavioural part and the ability to generate code in executable, industry-standard platforms.

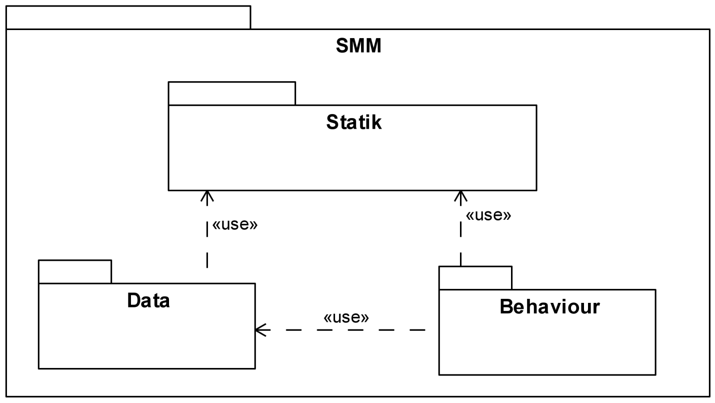


Figure 4.2: SMM Packages

The SMM has been built using the Eclipse Modelling Framework (EMF) introduced in chapter 2. It consists of 59 classifiers and 57 structural features. The meta-model is split into three packages (cf. figure 4.2) which are discussed in the following three subsections. Within the `SMM` package, the `Statik`[1] package defines the static aspects of a SOA (such as participants, partners, and message types). Based on these aspects, data handling statements (such as assignments and data manipulation actions) are defined in the `Data` package and, finally, the behaviour of SOA participants (i.e. the actual steps to be executed, and the protocols to be adhered to) in the `Behaviour` package.

4.2.1 Static Aspects

The `Statik` package of the SMM defines the basic static structure expected in a SOA and is shown in figure 4.3. The root entry point of an SMM model is the class `Participant`, which is shown in the top left.

A participant is a software artefact which provides or requires several services. Through these services, the participant addresses partners (or is addressed by a partner). The `Service` class is an abstraction of a service interaction point — either a service is provided by the participant (`ProvidedService`) or required (`RequiredService`).

As the root entity of an SMM model, the participant also contains a list of `SMMType`s. These types are used in interfaces and as message types for transmitting information.

Each service has a type (`InterfaceType`), which includes a set of implemented and used operations (bottom right). For example, a provided service (class `ProvidedService`) may contain both implemented operations (invoked on the participant) and used operations (invoked, as callbacks, on the partner). On the other hand, implemented operations of a `RequiredService` are invoked on a partner, while used operations are invoked as callbacks on the participant itself. As usual, an operation may contain input parameters and output parameters which are again typed with `SMMType`.

The central class `SMMType` has several subclasses for denoting different kinds of types. On the left, the supported `PrimitiveType` elements are shown. These types capture the primitive types encountered in most programming languages. For exchanging data between services, the `MessageType` element is used, which is a data-only class without any operations. It contains several `MessageProperty` elements which are used to model both primitive properties as well as associations. Finally, `ExceptionType` is used for modelling exceptions, and the special `NullType` represents a non-existent type of a variable.

Finally, note the class `TypedMultiElement`. This class captures multiplicities in associations: Message properties, operation parameters, and (later) variables may be used to store more than one element in an ordered or unordered way. This is captured through the `min`, `max`, and `ordered` properties.

[1]ending in a `k` due to `static` being a reserved keyword in Java.

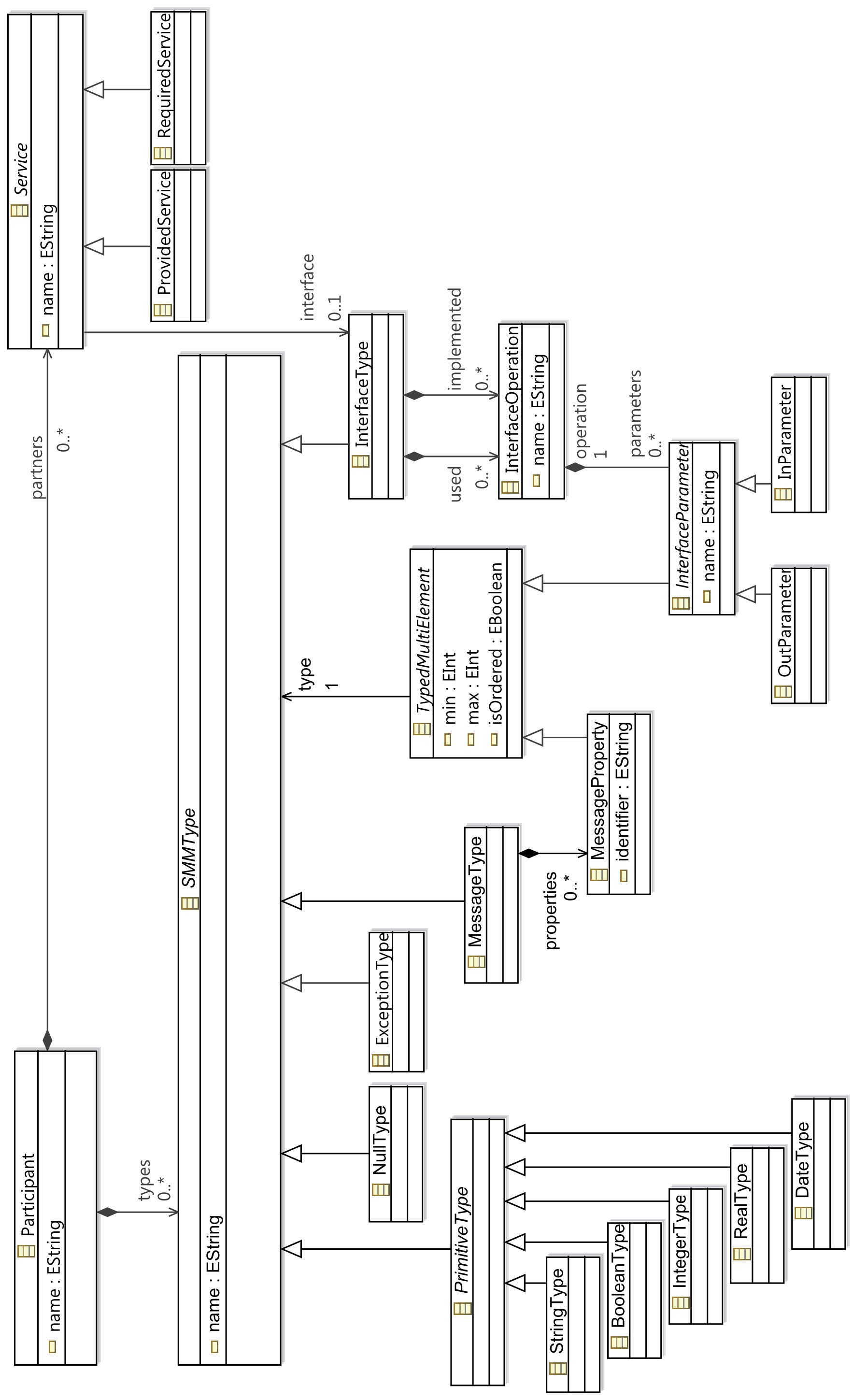


Figure 4.3: SMM: Statik Package

4.2.2 Handling Data

The SMM `Data` package defines the structures required for storing data expressions used in assignments, declarations, and parameters of service calls and call receptions. Furthermore, it includes support for defining and referencing variables and attributes within variables. The `Data` package is shown in figure 4.4.

The root class of the `Data` package is the `Expression` class. An SMM `Expression` has a type (imported from the `Statik` package) and is realised as either a `RightHandSideExpression` or a `LeftHandSideExpression`. These two classes match the usual idea of the left- and right-hand side of an assignment. The left-hand side represents a reference to a data container, such as a variable or a property, which is represented in the SMM by means of the `VariableReference` and the `PropertyReference` classes. A right-hand side may represent literals or operation calls in addition to data containers. The `Literal` class models literals in one of the primitive data types of the `Statik` package, while the `Operation` class models the invocation of a standard operation. The SMM defines these operations in the `OperationType` class:

- The first six operation types are mathematical operations for adding, subtracting, multiplying and dividing numbers as well as performing the modulo operation and negating numbers. The `add` operation is overloaded to also handle string concatenation.

- The other nine operations are used in conditions; the usual logical operations `and`, `or`, and `not` are used to combine statements, while `equals`, `greater`, `less`, and `not-equals` are used to form boolean expressions.

Right- and left-hand-side expressions are brought together with assignments as modelled by the `Assignment` class. Furthermore, variables can be declared with the `Declaration` class. Finally, the `InteractionParameter` class models either incoming or outgoing parameters of a service call. A `SendParameter` can carry a complete right-hand side expression which is to be sent to a partner, while the `ReceiveParameter` may only specify a left-hand side expression for storing the data.

4.2.3 Modelling Behaviour

The last package in the SMM meta-model is the `Behaviour` package, which is shown in the two figures 4.5 and 4.6. The `Behaviour` package includes elements from the `Data` and `Statik` package to define a structure for the execution of a participant, which includes service communication, structured elements like loops, parallel behaviour, or decisions, as well as the new service-oriented concepts of event- and compensation handling. Furthermore, it contains elements for defining protocols for services.

Figure 4.5 is concerned with the structuring actions and non-communication actions of the behaviour of a participant. On the bottom right, the `Participant` class from the `Statik` package is shown. The `Behaviour` package adds a new

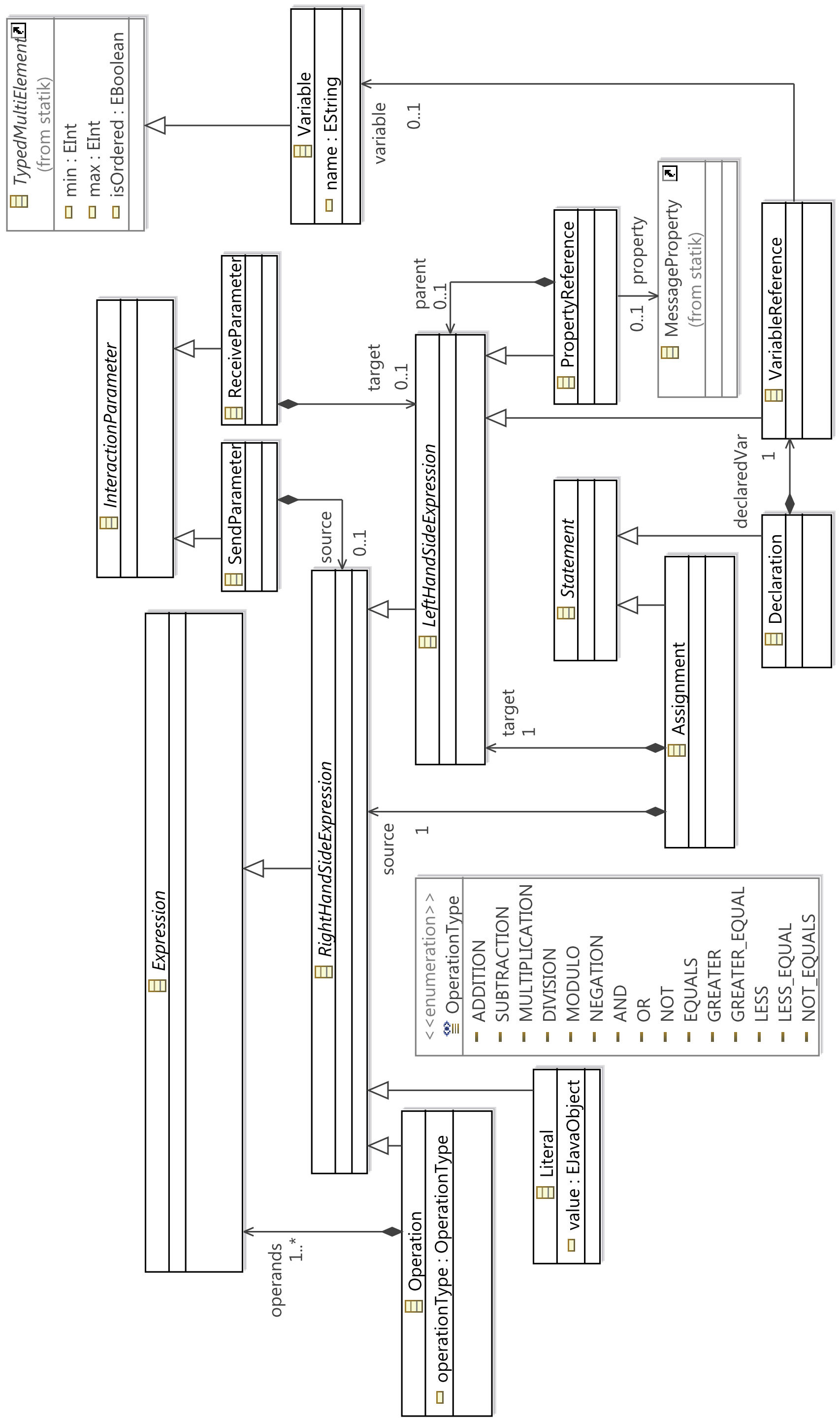


Figure 4.4: SMM: Data Package

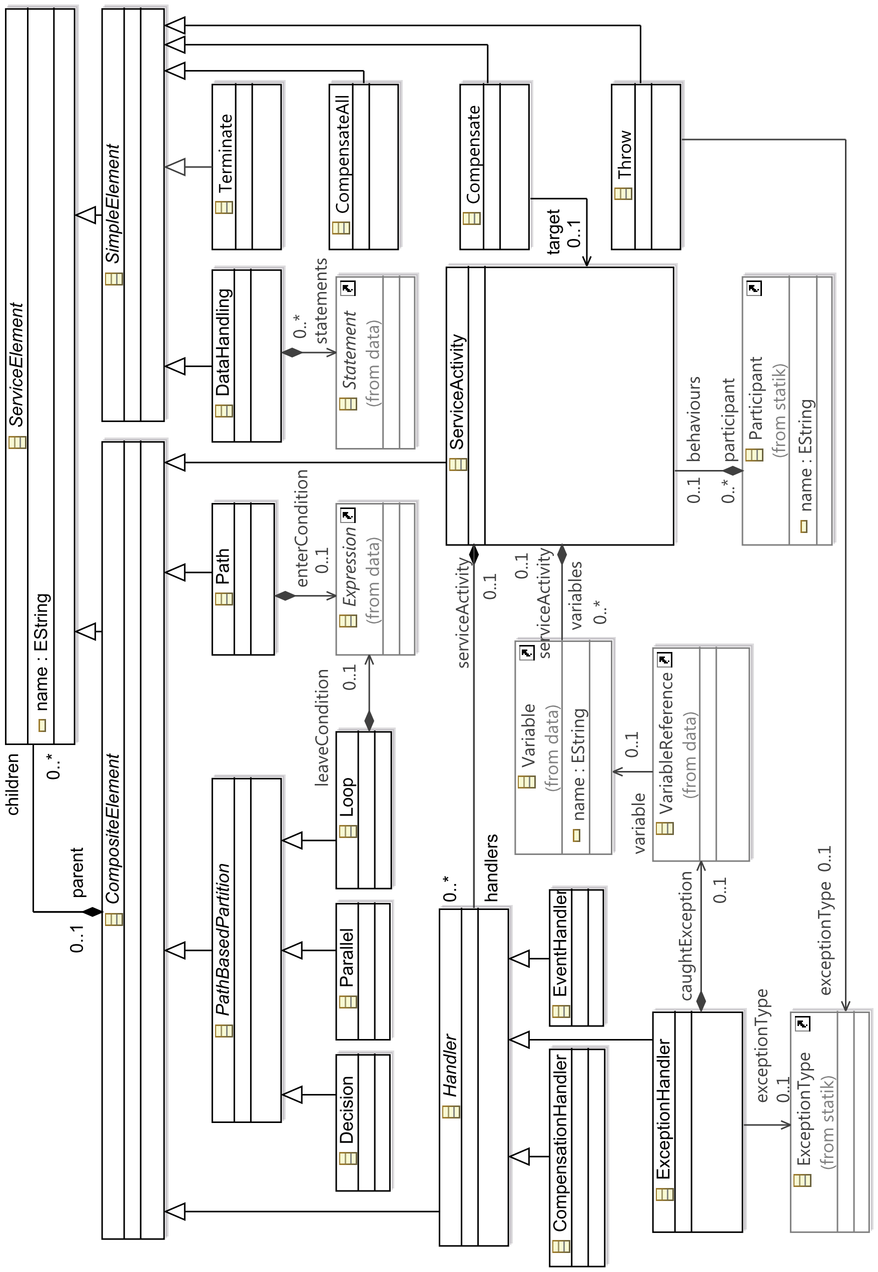


Figure 4.5: SMM: Behaviour Package (1/2)

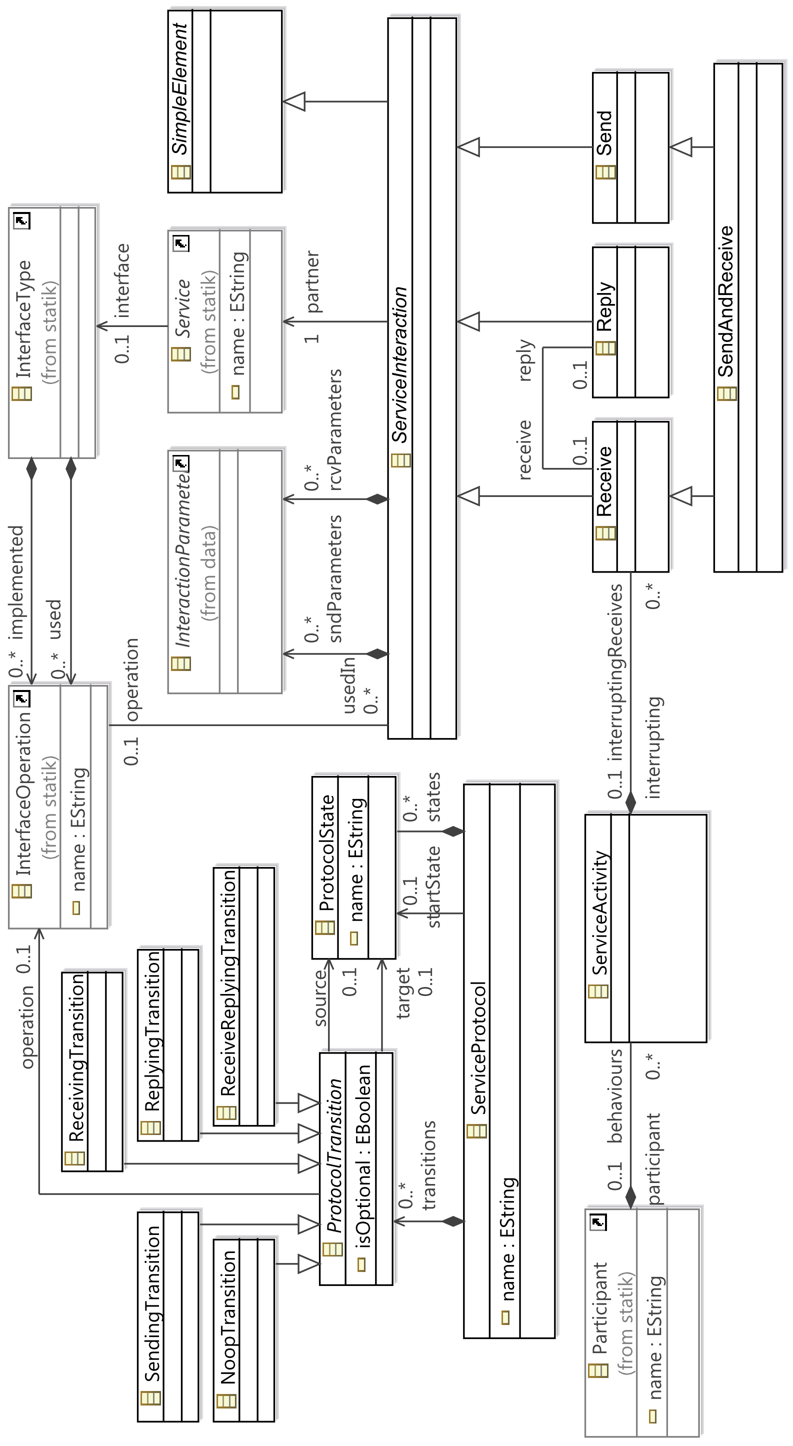


Figure 4.6: SMM: Behaviour Package (2/2)

association to this class, namely a list of ServiceActivity elements which, on this level, each represent one complete behaviour of the participant.

ServiceActivity is one of the two central classes in the Behaviour package (the other main class is ServiceProtocol, see below). It contains — transitively — all other elements associated to this behaviour of a participant. ServiceActivity is a specialisation of CompositeElement, which is in turn a specialisation of ServiceElement. CompositeElement, as the name suggests, follows the composite pattern, and may contain a number of children of type ServiceElement. In particular, if a CompositeElement contains more than one child, the children are assumed to be in sequence (executed one after another). Through this mechanism, the complete behaviour can be modelled, as all simple and structured actions are subclasses of ServiceElement — including ServiceActivity, which allows nesting of behaviour.

In its role as a structuring construct, ServiceActivity may contain a number of variables (Variable class from the Data package) and a number of event, exception, and compensation handlers. A ServiceActivity with an attached compensation handler may also be the target of a Compensate action (middle right).

For structuring child elements non-sequentially, the SMM includes three composite elements modelled as subclasses of PathBasedPartition, which models a partition in the execution workflow with a specialised meaning attached to its children, which must be instances of the type Path.

- A Decision models a choice: Just one of the paths is executed; which one depends on the attached enterCondition which is modelled as an Expression (from the Data package).
- A Parallel models concurrent behaviour of child paths.
- Finally, a loop indicates that the children in its (only) path are to be looped until the leaveCondition evaluates to true.

Subclasses of Handler already mentioned above are again composite elements which model compensation, exception and event handling, respectively.

- Exception handlers stand out as they both have a type of exceptions to be caught (SMMType class from the Statik package) as well as a place to store and reference the caught exception (VariableReference class from the Data package).
- Compensation handlers are used to model behaviour which *undoes* the work of the service activities they are attached to. A compensation handler may specify arbitrary elements; it is invoked by means of the Compensate or CompensateAll constructs (see below).
- Finally, event handlers are used to attach optional parallel behaviour to a ServiceActivity. An event handler may be executed at any time, and

also multiple times, during the execution of the attached `ServiceActivity`. It also may not be invoked at all. An event handler must begin with a `Receive` element (see below).

The right-hand side of the figure shows the non-communicating simple actions. All non-structuring actions are subclasses of `SimpleElement`. Two deal with compensation handling: The `Compensate` class models the compensation of a single referenced activity, while the `CompensateAll` class models default compensation for all nested service activities: By default, all successfully completed inner activities of the compensated `ServiceActivity` are compensated in reverse order of their execution.

The `Throw` element models an exception; the type of the exception is referenced in the `exceptionType` association (by means of `ExceptionType` from the `Data` package). `Terminate` ends the complete behaviour. Lastly, the `DataHandling` element may contain a number of data manipulation statements which are modelled by reference to the `Statement` class from the `Data` package.

The second figure (figure 4.6) is concerned with the communication actions of a participants behaviour as well as the protocols of services.

The right-hand side of the figure shows the communication actions, i.e., `ServiceInteraction` elements (middle right), of which there are four:

- `Receive`, already mentioned above, waits for an incoming call from a partner of the participant. A receive may be an interrupting receive, it which case the link to a corresponding parent service activity is set; otherwise, this link is left open. An interrupting receive can end the activity at any time.

- `Send` sends out a call to a partner without waiting for a return value.

- `Reply` has the same semantics as `Send`, but ends a previous call from a partner and, for this reason, contains a link to the previous `Receive`.

- `SendAndReceive` combines a `Send` with a `Receive`, i.e. an operation call is sent out and an answer is received.

The `ServiceInteraction` class contains a set of links which are (mostly) common to its subclasses. First of all, a service interaction references a service to which a call is sent, or from which a call is received (`partner` association, referencing `Service` from the `Statik` package). The operation which is expected or invoked is modelled by means of the `operation` association, referencing the `InterfaceOperation` class from the `Statik` package. Lastly, an interaction may contain a set of parameters to be sent or received; depending on the concrete subclass of `ServiceInteraction`, only one association or both may be used. Both parameter sets are referenced as `InteractionParameter` elements from the `Data` package.

Finally, the left-hand side of this figure contains the elements for protocols. Each `InterfaceType` (from the `Statik` package) may have an associated

`ServiceProtocol`, which is — in effect — a state machine consisting of `ProtocolState`s and `ProtocolTransition`s linking them together; the protocol itself has a designed start state. There are five `ProtocolTransition` subtypes:

- `SendTransition` indicates that the participant to which the protocol is attached to sends a call.
- `ReceivingTransition` indicates that the participant to which the protocol is attached to receives a call.
- `ReplyTransition` denotes the fact that a reply is being sent out to a previous receive.
- `ReceiveReplyTransition` denotes the fact that a reply is *expected* from the behaviour to which the protocol is attached.
- Finally, `NoopTransition` is a transition without any externally visible effect (i.e., no communication).

The operation involved is given as an association to `InterfaceOperation` (from the `Statik` package). Note that a transition may be optional, as indicated by the `isOptional` flag.

This concludes the discussion of the SMM meta-model for service-oriented architectures. The next section will discuss a textual syntax for the behavioural part, which will become important in chapter 5.

4.2.4 Textual SMM

The SMM, as a meta-model, does not include any syntax; this part is played by the UML, SoaML, and UML4SOA as shown in the last chapter. However, for the definition of the formal semantics using modal I/O automata in chapter 5, it is convenient to have a textual notation available for the parts of the SMM relevant to the semantics, and which benefit from a textual notation. In our case, this is the definition of the behaviour of a SOA participant. A textual notation for protocols is not required as the translation is straightforward (cf. chapter 5).

We can take advantage of the well-nestedness of the SMM during the definition of the syntax; the resulting code is well-nested, too.

4.2.4.1 SMM Behavioural Textual Syntax

As in chapter 3, we again use the syntax definition style from the Java language specification. We start with a number of definitions for identifiers.

Listing 4.1: SMM Syntax: Identifiers

```
OperationIdent:
ExceptionIdent:
ActivityIdent:
    String
```

We now define the denotation of an SMM participant behaviour, which is an SMM `ServiceActivity`. An activity definition begins with the keyword **activity**, has a name, and must contain an inner *Element*. It may have, additionally and optionally, associated exception, event, and compensation handlers as well as interruptions, mirroring the meta-class associations in the SMM.

Listing 4.2: SMM Syntax: Services

```
ServiceActivity:
    activity(ActivityIdent : Element ActivityInterruptions_opt)
        ActivityExceptions_opt ActivityEvents_opt ActivityCompensation_opt
```

Before presenting the child elements, we introduce interruptions and handlers. Firstly, the SMM defines the concept of interrupting receives, allowing the interruption of a `ServiceActivity`. An interrupting receive is modelled with the `Receive` meta-class which has its `interrupting` association set to a `ServiceActivity`; in turn, the `ServiceActivity` references the `Receive` in its `interruptingReceives` association. We denote an interrupting receive with the keyword **interrupt** and a *Receive* element (to be defined later).

Listing 4.3: SMM Syntax: Interruptions

```
ActivityInterruptions:
    ActivityInterruption
    ActivityInterruptions ActivityInterruption

ActivityInterruption:
    interrupt Receive
```

Exception handlers, which are modelled in the SMM with the `ExceptionHandler` meta-class, are defined with the keyword **exception**, an exception to catch, and a body. The exception to catch corresponds to the `exceptionType` association of the `ExceptionHandler` class and thus corresponds to an `ExceptionType`. There may be multiple exception handlers attached to a service activity.

Listing 4.4: SMM Syntax: Exception Handlers

```
ActivityExceptions:
    ActivityException
    ActivityExceptions ActivityException

ActivityException:
    exception ExceptionIdent : Element
```

Event handlers (`EventHandler` meta-class) are defined by using the **event** keyword. They do not need any additional information — the SMM requires the first element of an event handler to be a *receive* action, though.

Listing 4.5: SMM Syntax: Event Handlers

```
ActivityEvents:
    ActivityEvent
    ActivityEvents ActivityEvent

ActivityEvent:
    event Element
```

Finally, a service activity may have one compensation handler (`CompensationHandler` meta-class) which is denoted with the keyword **compensation**.

Listing 4.6: SMM Syntax: Compensation

```
ActivityCompensation:
    compensation Element
```

We have now bootstrapped the definition of SMM participant behaviours. Each service activity as well as all handlers are SMM `CompositeElement`s, which may contain several children. We allow the notation of possible children with the *Element* non-terminal.

Listing 4.7: SMM Syntax: Elements

```
Element:
    Activity
    StructuredElement
    BasicAction
```

As can be seen from this definition, an *Element* can (again) be an activity, a structured element, or a basic action (send, receive, etc.). Most commonly, an activity will contain a structured element as its immediate child, which is defined as follows:

Listing 4.8: SMM Syntax: Structured Elements

StructuredElement:
Sequential
Parallel
Decision
Loop

These elements correspond to the basic, well-nested structuring mechanisms in the SMM. A sequential block simply consists of a number of statements separated by semicolons (**;**); this notation is used for SMM `CompositeElement`s with more than one child. A parallel block (meta-class `Parallel`, denoted as **parallel**) consists of multiple individual elements which all run in parallel. In the case of a decision (`Decision` meta-class, denoted as **decision**), one of the elements is picked for execution. Finally, a loop (`Loop` meta-class, denoted as **loop**) indicates that the enclosed element is to be repeated.

Listing 4.9: SMM Syntax: Sequential, Parallel, Decision and Loop

Sequential:
Element
Sequential **;** *Element*

Parallel:
parallel(*Element* | *Element***)**

Decision:
decision(*Element* | *Element* **)**

Loop:
loop(*Element***)**

Leaving the area of structured statements, we now move on to the discussion of actions. The abstract non-terminal for actions is *BasicAction*, which is split into *ServiceAction* and *ControlAction*:

Listing 4.10: SMM Syntax: Actions

BasicAction:
ServiceAction
ControlAction

*ServiceAction*s are used for communicating with other services from within the participant; *ControlAction*s trigger exceptions and compensations.

Listing 4.11: SMM Syntax: ServiceActions and ControlActions

```
ServiceAction:
    Send
    Receive
    Reply
    Send&Receive

ControlAction:
    Throw
    Compensate
    CompensateAll
```

Finally, we can specify the terminals for these actions. The names of these terminals directly map to the corresponding SMM meta-classes (for example, the **send** terminal corresponds to the `Send` meta-class). Note that the *Data* action is missing from this list, as data handling is ignored in the formal semantics.

Listing 4.12: SMM Syntax: Action Terminals

```
Send:
    send(OperationIdent)

Receive:
    receive(OperationIdent)

Reply:
    reply(OperationIdent)

Send&Receive:
    send&receive(OperationIdent)

Throw:
    throw(ExceptionIdent)

Compensate:
    compensate(ActivityIdent)

CompensateAll:
    compensate()
```

4.2.4.2 Example

As an example, we refer back to the ThesisManager process from the case study already modelled in chapter 3 and directly translate the sub-activity *registration* shown in figure 4.7 into the textual syntax of the SMM (listing 4.13).

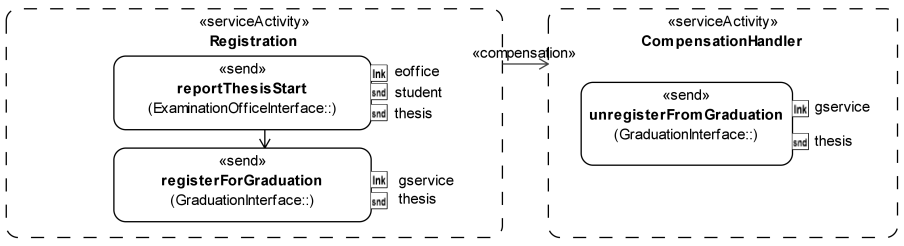


Figure 4.7: eUniversity Case Study: Registration Activity (Graphical)

A complete description of the parsing of UML4SOA models to the SMM will be given in chapter 6.

Listing 4.13: eUniversity Case Study: Registration Activity (Textual)

```
activity (Registration :
      send( reportThesisStart ) ;
      send( registerForGraduation )
) compensation activity (CompensationHandler :
      send( unregisterFromGraduation )
)
```

4.3 Summary

This chapter has introduced the Service Meta-Model (SMM), which forms the underlying basis of the three main components of the MDD4SOA approach — modelling, analysis, and code generation.

The SMM is a generic, language- and platform-independent meta-model for describing the static, dynamic, and data handling aspects of a SOA system. We have first given an overview of how the SMM fits into the MDD4SOA approach (section 4.1), followed by a description of the meta-model itself (section 4.2). The latter has included a textual notation for the behavioural part of the SMM.

The UML, SoaML, and the UML4SOA profile as defined by the UML4-SOA/Strict dialect form the concrete syntax of the SMM. A formal semantics and analysis methods for the behavioural parts of the SMM will be discussed in the next chapter.

Finally, the model transformations for code generation introduced in chapter 6 use the complete SMM as the source model. This chapter also includes a detailed description of how to parse a combined UML, SoaML, and UML4SOA model into an instance of the SMM.

Chapter 5

Semantics and Analysis

Chapters 3 and 4 have introduced the UML4SOA profile and the Service Meta-Model (SMM), both for participant behaviours and protocols, along with a natural language description of the *meaning* of the newly introduced constructs. However, for the purpose of analysis and verification, a more rigorous description of these models is required.

In this chapter, we therefore introduce a formal semantics for UML4SOA participant behaviours and protocols (via the Service Meta-Model). As the semantic domain, we employ *modal input/output transition systems (MIOs)*, which have already been introduced in chapter 2 (section 2.5). The mapping between SMM models and MIOs is given in a *denotational style.* The mapping function is fully executable and tool-supported.

A formal semantics as the one introduced here forms the basis for rigorous analysis and verification of the modelled system. As outlined in chapter 2, *interface theories* can be used for formally specifying and verifying notions of interface conformity and compatibility. In addition to existing interface theories which are directly usable on the formal semantics of UML4SOA, we also introduce a domain-specific interface theory for checking *observational* compliance.

To aid in the verification of UML4SOA models in their MIO representation, we have furthermore created a verification tool, the Mio Workbench, which supports verification of MIOs using a variety of interface theories and their notions of refinement and compatibility.

This chapter is structured as follows. We introduce the denotational semantics for UML4SOA in section 5.1. Formal analysis based on MIOs is discussed in section 5.2. Tool support, which includes the executable semantics, the Mio Workbench, and a tool for back-annotation of analysis results to UML, is examined in section 5.3. Related work is presented in section 5.4. We conclude this chapter with a summary in section 5.5.

Published results: Results presented in this chapter are based on publications [SM08], [BMSH10], and [MSB10].

5.1 A Formal Semantics for UML4SOA

This section is concerned with defining a formal semantics for the participant behaviours and service protocols of UML4SOA. The semantics is defined on the Service Meta-Model introduced in the previous chapter; as UML4SOA models form the syntax of the SMM, the semantics is defined for and can be (mechanically) produced from both.

We follow a denotational style to devise the semantics of the SMM, i.e., we specify a semantic function which maps the elements of the SMM to mathematical objects. We employ the semantic domain of modal input/output automata (MIOs). MIOs are a good match for the SMM models as they:

- allow the specification in terms of a transition system, which captures the workflow-like ideas behind the SMM,
- natively support input and output actions, matching the send and receive operations introduced in the SMM,
- can distinguish between *required* and *optional* operations. Optional (*may*) transitions in protocols are required to be able to verify optional implementation behaviour such as compensation calls.

Translating SMM behaviours into MIOs needs to take the SOA elements of compensation, events, and exceptions into consideration and thus requires a complex transformation process. This process is thus discussed first in the next section. SMM protocols, on the other hand, are already state machines and are thus easier to translate. They will be dealt with in the subsequent section.

5.1.1 SMM Participant Behaviours

The semantics presented in this chapter is inspired by the semantics given for FSP in [MK99]. The particular focus of the semantics for the SMM lies on the newly introduced elements for service-oriented computing: communication, compensation handling, event handling, and the interactions of each of these with exceptions. Thus, we disregard the data types and the data access and manipulation functions of the SMM.

Note that some parts of the denotational function are specified in an imperative, algorithmic style. We have chosen this approach as the resulting algorithms are shorter and thus more readable and understandable. Furthermore, we only use local data inside the individual function definitions. The algorithms thus have an obvious translation into a purely functional style.

In the following, we will detail the formal semantics for SMM `ServiceActivity` elements. We use the textual notation defined in chapter 4 as a convenient shortcut to the SMM meta-model.

5.1.1.1 Preliminaries

Although the structure of SMM behavioural models is simple and easy to understand, the interactions of events, compensations, and exceptions are non-trivial. These interactions are inherent to services, and the translation function to MIOs has to reflect this complexity. Nevertheless, we have opted to present the complete semantics here as we feel that it is important to understand the above-mentioned interactions.

The translation is defined with a denotational-style semantics. We use a mapping function from syntax to semantics, which is defined on all non-terminals in the SMM syntax. For each non-terminal, we precisely define the mapping with a pseudo-code notation.

Furthermore, the definition of the algorithms can be greatly simplified by introducing an intermediate automaton which contains more information than the resulting MIO. This information is only used during the translation process and is not required anymore after the translation is finished, thus the intermediate automaton is mapped to a MIO as the last step in the process. Recalling from chapter 2, we can write a MIO as follows:

Listing 5.1: MIO Definition

```
MIO =< $states$,
       $startState$,
       $actions$,
       $transitions$ >

with $actions = \bigcup\{inputActions, outputActions, internalActions\}$,
     $transitions = \bigcup\{mustTransitions, mayTransitions\}$
```

For the transformation, we introduce an intermediate automaton (IA). This automaton carries some additional information detailed below, but disregards the difference between must and may transitions as we consider SMM service behaviour to be implementations in the terminology of MIOs.

Listing 5.2: IA Definition

```
IA  =< $states$,
       $startState$,
       $endStates$,
       $actions$,
       $transitions$,
       $comps$ >

with $endStates = \bigcup\{eeStates, neStates\}$,
     $actions = \bigcup\{inActions, outActions, intActions\}$
```

The elements introduced to the IA — *eeStates*, *neStates*, and *comps* — have the following intuition:

- *eeStates* (error end states) and *neStates* (normal, or *non-error*, end states) are two disjunct sets of states representing the end states of the automaton. The states are also part of the *states* set of the IA.
- $comps : String \rightarrow IA$ is a function for storing compensation information. *comps* maps activity names to the actual automaton which was created for the compensation handler of the activity indicated by the given name.

In the following, we define a function $ia[\![]\!]$ with the following signature:

$$ia : \text{SMMNonterminal} \rightarrow IA$$

$ia[\![]\!]$ is defined on all non-terminals of the SMM textual syntax introduced in chapter 4. This function yields an IA automaton. An IA can be mapped to a MIO by disregarding (*forgetting*) the additional information and using the transition set as both must and may transitions. This is done with the function $forget : IA \rightarrow MIO$:

Listing 5.3: Mapping IAs to MIOs

$$\begin{aligned}&forget(< states, start, ends, actions, transitions, comps >) = \\ &\qquad < states, start, actions, transitions >\end{aligned}$$

As the notation for both MIOs and IAs, we use the standard labelled transition system syntax:

- A *state* is denoted by a white circle with a solid line around it. The start state of an automaton is denoted as a grey circle.
- A *transition* is denoted by a directed edge between states with the transition label written alongside the transition. Output labels are suffixed with an exclamation mark (!); input labels with a question mark (?). Internal labels are suffixed with a semicolon (;). We leave these suffixes out if they are not relevant.

MIOs have two different sets of transitions, namely must- and may transitions. The former are denoted by a solid line, the latter by a dashed line. However, as an IA does not differentiate between these two types, we will mostly see solid lines.

For IAs, we add the following extensions:

- *End states* are denoted by black circles instead of white ones.
- The names of installed *compensation handlers* are noted as text beneath the state.

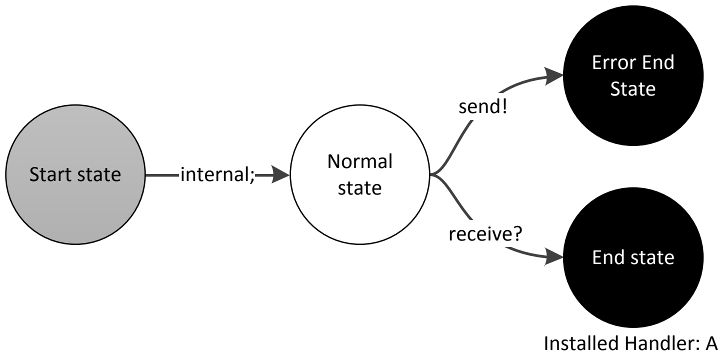


Figure 5.1: Notation for IAs

An example for an IA is shown in figure 5.1. To reach the corresponding MIO of an IA, the information about end states and compensation installations is removed — no further changes are necessary.

Before we start with the definition of the function $ia[\![]\!]$, we first discuss the algorithm syntax, notations, and helper functions used in the following sections.

Algorithm Listings

In general, we denote each (helper) function with an algorithm. An example is shown in listing 5.4.

Listing 5.4: Algorithm Example

```
Definition
    Signature

Input/Output
    Input and Output

Algorithm
    $\text{Statement}_1$
    ...
    $\text{Statement}_n$
```

Each algorithm is split into three parts:

- **Definition**. The definition part denotes the function name, parameters, and the result. In most cases, the result will be an automaton which will be built up in the remainder of the listing.
- **Input and Output**. The input and output parts define how the parameters of the function are to be used in the algorithm as well as the result

of the algorithm.

- **Algorithm**. The algorithm part contains a pseudo-code listing of how the returned element of the function is built.

Notations

In the algorithm section, we use a set of notations for precisely and concisely noting the operations performed. First, we define a set of assignment operators.

- $\leftarrow$ is the normal assignment operator: The right-hand side is assigned to the left-hand side.

- $\leftarrow_{add}$ *adds* the single element or the set of elements given on the right-hand side into the set given on the left-hand side.

- $\mapsto$ adds a new element to a specified function, thus defining the mapping of one element of the domain to one element of the range.

We denote a transition as a three-tuple $(startState, \underline{transition}, endState)$. Adding such a transition to an automaton has the double effect of a) adding the transition and b) adding the underlined transition label if it does not yet exist. To distinguish between the three sets of labels present in the automaton, we use the symbols ?, !, and ; to indicate the set of input, output, and internal actions, respectively. These symbols are not part of the transition label.

Helper Functions

Finally, we define a set of helper functions required to describe the algorithms without too much technical detail. The reader may refer back to this list as required when reading the algorithms.

- $addAll : IA \rightarrow IA$ *adds* elements from the automaton specified on the right-hand side to the automaton specified on the left-hand side. It is important to note that elements already existing in the automaton on the left are not removed.

- $addPreSpacing : IA \times State \rightarrow IA$ and $addPostSpacing : IA \times State \rightarrow IA$ returns a new version of the given automaton, in which an *interspace* internal transition is placed before or after, respectively, the given state. The given state must be a start or end state; thus, to add the internal transition, a new start state or a new end state is introduced. The new automaton includes all states, transitions, actions, and compensation installations of the old one plus the new state and transition, which have not been in the automaton before.

  Interspaces are needed to prevent situations such as consecutively adding two loops and being able to execute the first one again after the second.

- $cInstallTransitions : IA \times State \rightarrow \wp(Transition)$ returns the set of transitions where the transition label starts with *compInstalled*, starting backwards from the given transition to the start state of the automaton — unless there is a transition in-between with a *compHandled* label (in which case the handler has already been executed).

- $cCallTransitions : IA \rightarrow \wp(Transition)$ returns all transitions in the given automaton where the transition label starts with *compensate*.

- $compensationsMatch : Transition \times Transition \rightarrow Boolean$ returns true if the label of the first transition starts with a label *compensate* or *compensateAll* and the label of the second transition starts with a label *compInstalled*. In case the first transition models a *compensateAll*, true is returned. Otherwise, true is returned if the service activity given after *compensate* equals the one given after *compensateAll*.

- $actName : Transition \rightarrow String$ returns an activity name as found in the label of the given compensation-related transition. We use three compensation label types for denoting the installation of a compensation handler (*compInstalled(activityName)*), the usage (or uninstallation) of a compensation handler (*compHandled(activityName)*), and finally the call for invoking a compensation handler (*compensate(activityName)* or simply *compensateAll*).

- $newState : IA \rightarrow IA \times State$ returns a new version automaton with a new state $State$.

- $exceptionsMatch : \wp(Transition) \times ident \rightarrow Boolean$, where $state$ is a non-error end state, $ident$ an exception identifier, and IA an automaton returns true if one of the given transitions has a label which starts with *throw* and contains the $ident$ exception identifier.

- $isErrEnd : IA \times State \rightarrow Boolean$ and $isNormEnd : IA \times State \rightarrow Boolean$ return $true$ or $false$, depending on whether the given state is in the set of error ends or non-error ends.

- $mergeEnds : IA \rightarrow IA$ returns a new version of the given automaton in which all non-error end states of the given automaton which share the same set of installed compensation handlers (given by $cInstalls$) are merged.

- $mergeStates : IA \times State_1 \times \cdots \times State_n \rightarrow IA \times State$, where $State_1 \ldots State_n$ are states from $states_{IA}$, returns a new version of the automaton in which the given states are merged into one. The result is that each incoming edge of the given states now leads to the new state, and all outgoing edges of the given states start from the new state. The original states are no longer present. The result of this function is the new IA and the newly merged state.

- $relabel : IA \rightarrow IA$ returns a relabelled automaton.

- $\mathit{isInLoop} : IA \times Transition \rightarrow \mathit{Boolean}$ returns true if the $Transition$ is in a loop within IA, which means that it is possible to return to $Transition$ through the target state.
- Additionally, we use the following convenience functions for working with IAs:
  - $\mathit{out} : IA \times \mathit{State} \rightarrow \wp(Transition)$ returns the outward transitions in IA from state State,
  - $\mathit{in} : IA \times \mathit{State} \rightarrow \wp(Transition)$ returns the inward transitions in IA to state State,
  - $\mathit{label} : Transition \rightarrow \mathit{String}$ returns the label of the transition $Transition$ in IA,
  - $target : Transition \rightarrow \mathit{State}$ returns the target state of the transition in IA, and
  - $\mathit{source} : Transition \rightarrow \mathit{State}$ returns the source state of the transition in IA.

5.1.1.2 Service Actions

Notation

Service Actions are denoted as follows:

send(*OperationIdent* **)**
receive(*OperationIdent* **)**
reply(*OperationIdent* **)**
send&receive(*OperationIdent* **)**

Intuition

A send, receive, reply, or send&receive in the SMM is an action within an activity. In an automaton, an action is not attached to a state, but to a transition. We use the following mapping from SMM actions to transitions:

SMM	MIO
Send	Output
Receive	Input
Reply	Output
Send & Receive	Two transitions: Input, Output

Table 5.1: Mapping of SMM Actions to MIO Transitions

Each send, receive, or reply is therefore a transition in the automaton. In case of send&receive, we get two transitions.

The first non-terminal to be converted is the *send* action of the SMM, which models an operation call from the participant to an outside partner. A *send* is translated with the following algorithm.

Listing 5.5: MIO: Send

```
Definition
    $ia[\![send(opName)]\!] = S$

Algorithm
    $\text{states}_S \leftarrow \{\ \text{startSend, endSend}\ \}$
    $\text{startState}_S \leftarrow \text{startSend}$
    $\text{neStates}_S \leftarrow \{\ \text{endSend}\ \}$
    $\text{outActions}_S \leftarrow \{\ \text{opName}\ \}$
    $\text{transitions}_S \leftarrow \{\ (\text{startSend},\underline{\text{opName!}},\text{endSend})\ \}$
```

The definition shows that the *ia* function is now defined on the SMM textual syntax non-terminal $send(opName)$, and yields an intermediate automaton (IA) S. S contains two states: *startSend* and *endSend*, of which *startSend* is the start state and *endSend* is an end state. Furthermore, the IA contains one output action $opName$, and one transition from *startSend* to *endSend* which is labelled with the output action $opName$.

As an initial example, figure 5.2 shows how the UML4SOA concrete notation maps into an IA. On the left, a send action in UML4SOA is shown; on the right, the same action as an IA. In the IA, *startSend* is the start state, *endSend* a (normal) end state. The transition in-between has the output label *reportThesisStart*. As mentioned above, the corresponding MIO to the IA looks the same, but is without a special denotation of the end state.

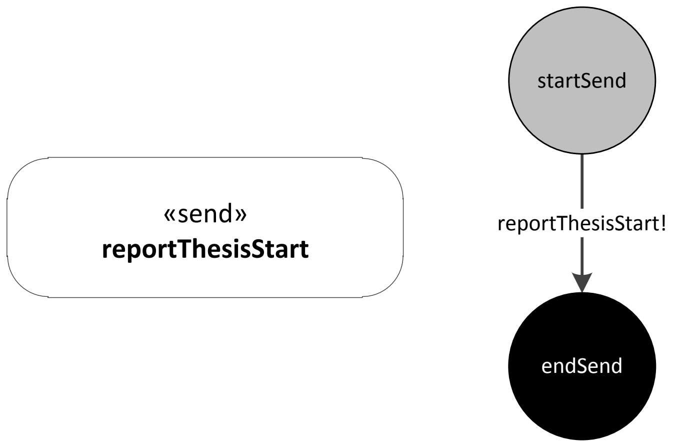


Figure 5.2: Example: A Send in UML4SOA and as an IA

The remaining communication actions — *receive*, *reply*, and *send&receive* — are converted likewise. For *receive*, we get:

Listing 5.6: MIO: Receive

```
Definition
    $ia[\![receive(opName)]\!] = S$

Algorithm
    $\text{states}_S \leftarrow \{$ startRcv, endRcv $\}$
    $\text{startState}_S \leftarrow$ startRcv
    $\text{neStates}_S \leftarrow \{$ endRcv $\}$
    $\text{inActions}_S \leftarrow \{$ opName $\}$
    $\text{transitions}_S \leftarrow \{$ (startRcv,$\underline{\text{opName?}}$,endRcv) $\}$
```

A *reply* is denoted similar to a send; however, we add a `return_` prefix indicating that this output action is in response to a previous input action:

Listing 5.7: MIO: Reply

```
Definition
    $ia[\![reply(opName)]\!] = S$

Algorithm
    $\text{states}_S \leftarrow \{$ startReply, endReply $\}$
    $\text{startState}_S \leftarrow$ startReply
    $\text{neStates}_S \leftarrow \{$ endReply $\}$
    $\text{outActions}_S \leftarrow \{$ "return_" + opName $\}$
    $\text{transitions}_S \leftarrow \{$ (startReply,$\underline{\text{"return\_" + opName!}}$,endReply) $\}$
```

Finally, *send&receive* uses two transitions, one for send and one for receive. Again, we use a prefix `return_` for denoting that the input action is an expected return of the previous output action.

Listing 5.8: MIO: Send&Receive

```
Definition
    $ia[\![send\&receive(opName)]\!] = S$

Algorithm
    $\text{states}_S \leftarrow \{$ startS&R, midS&R, endS&R $\}$
    $\text{startState}_S \leftarrow$ startS&R
    $\text{neStates}_S \leftarrow \{$ endS&R $\}$
    $\text{inActions}_S \leftarrow \{$ "return_" + opName $\}$
    $\text{outActions}_S \leftarrow \{$ opName $\}$
    $\text{transitions}_S \leftarrow \{$ (startS&R,$\underline{\text{opName!}}$,midS&R),
                                   (midS&R, $\underline{\text{"return\_" + opName?}}$, endS&R) $\}$
```

5.1.1.3 Control Actions

The actions *throw*, *compensateAll* and *compensate* are used for controlling the flow in an activity. We convert these actions to transitions in the automaton. All three actions cannot be given meaning considering only the action itself — instead, meaning is given to them later by considering the environment — in particular, an enclosing activity.

Throw

Notation

throw(*ExceptionIdent***)**

Intuition

A *throw* action marks an exception, and thus the end of a path — the path is not continued. When encountered, a throw transition *directly* leads to a final state which is called an *error end state*. When we later handle a complete service activity which has an exception handler, we can revisit these error states and, if the exception handler matches the exception thrown, attach the handler to the error state to handle the exception, thus creating a *non-error end state*.

Note that an exception handler may not be encountered in the direct parent activity of a *throw*, but also transitively in another parent. This is not a problem as the error end state stays untouched until a valid exception handler is found. If none is found at all, the error end state ends the complete automaton.

Listing 5.9: MIO: Throw

```
Definition
    $ia[\![throw(exName)]\!] = S$

Algorithm
    $\text{states}_S \leftarrow \{ \text{ startThrow, endThrow } \}$
    $\text{startState}_S \leftarrow \text{startThrow}$
    $\text{eeStates}_S \leftarrow \{ \text{ endThrow } \}$
    $\text{intActions}_S \leftarrow \{ \text{ throw(exName) } \}$
    $\text{transitions}_S \leftarrow \{ \text{ (startThrow,}\underline{\text{throw(exName);}}\text{,endThrow) } \}$
```

Compensation

Notation

compensate(*ActivityIdent***)**
compensateAll()

Intuition

A *compensate* or *compensateAll* action may only be used in an exception or compensation handler. A *compensate* or *compensateAll* marks the invocation of one or many compensation handlers, which are defined in inner activities of the activity the handler in which the *compensate* or *compensateAll* action is included is attached to.

When we encounter a *compensate* or *compensateAll* call, we are currently dealing with a *handler*, i.e. a service activity (see below). This means that we do not yet have access to the activities to compensate. Therefore, like in a *throw*, we note the fact that there was a *compensate* call and use this fact later when handling the service activity which the handler is attached to.

Listing 5.10: MIO: Compensate

```
Definition
    $ia[\![compensate(actName)]\!] = S$

Algorithm
    states$_S$ $\leftarrow$ { startComp, endComp }
    startState$_S$ $\leftarrow$ startComp
    neStates$_S$ $\leftarrow$ { endComp }
    intActions$_S$ $\leftarrow$ { compensate(actName) }
    transitions$_S$ $\leftarrow$ { (startComp,$\underline{\text{compensate(actName);}}$,endComp) }
```

Listing 5.11: MIO: CompensateAll

```
Definition
    $ia[\![compensateAll()]\!] = S$

Algorithm
    states$_S$ $\leftarrow$ { startComp, endComp }
    startState$_S$ $\leftarrow$ startComp
    neStates$_S$ $\leftarrow$ { endComp }
    intActions$_S$ $\leftarrow$ { compensateAll() }
    transitions$_S$ $\leftarrow$ { (startComp,$\underline{\text{compensateAll();}}$,endComp) }
```

5.1.1.4 Structured Actions

Structured actions are used to aggregate activities in a certain semantic way; we distinguish between the sequential operation, decisions, loops, and parallel execution.

An important challenge when handling structured actions is keeping *throw* and *compensation-installation information* in a correct way through the structured actions.

Sequential

Notation

$Element_1$ **;** $Element_2$

Intuition

Sequential execution of activities means concatenating their automata together, one after the other. In this process, *error end states* are ignored, because they mean that the whole sequence is aborted. A *non-error end state*, however, corresponds to a set of installed compensation handlers; we must keep this information throughout the sequential area. Therefore, when we encounter multiple non-error end states in an automaton, we *multiply* the automaton which comes after it.

Note that it might be necessary to add an interspace transition in-between two consecutively executed automaton in case the end states of the former have outgoing and the start state of the latter has incoming transitions. This ensures sequential execution.

The algorithm for sequential behaviour is shown in listing 5.12.

Listing 5.12: MIO: Sequential

```
Definition
    $ia[\![Element_1; Element_2]\!] = S$

Input/Output
    Let $A_1 = relabel(ia[\![Element_1]\!])$
    Let $A_2 = relabel(ia[\![Element_2]\!])$
    Let $S = \emptyset$

Algorithm
    $S \leftarrow$ addAll(states$_{A_1}$, $\emptyset$, <$\emptyset$, eeStates$_{A_1}$>, actions$_{A_1}$, transitions$_{A_1}$,
            comps$_{A_1}$)
    for each endState in neStates$_{A_1}$ do
        A$_{copy} \leftarrow$ relabel(A$_2$)
        $S \leftarrow$ addAll( states$_{A_{copy}}$, <$\emptyset$, eeStates$_{A_{copy}}$>, actions$_{A_{copy}}$,
                transitions$_{A_{copy}}$, comps$_{A_{copy}}$)
        if out(endState) $\neq \emptyset$ and in(startState$_{A_{copy}}$) $\neq \emptyset$ then
            S $\leftarrow$ addPostSpacing(S, endState)
        end if
        < S , _ > $\leftarrow$ mergeStates(S, endState, startState$_{A_{copy}}$)
        neStates$_S \leftarrow_{add}$ neStates$_{A_{copy}}$
    end for
    S $\leftarrow$ mergeEnds(S)
```

Decision

Notation

decision($Element_1 \mid Element_2$**)**

Intuition

Decision means only one of two possible automatons is executed. Each of the automata may have multiple non-error end states; if they contain the same set of compensation handler installations, they can be merged; otherwise, we just leave them as is. Error end states are, as usual, not handled.

The algorithm for decisions is shown in listing 5.13.

Listing 5.13: MIO: Decision

```
Definition
    $ia[\![\textbf{decision(}Element_1; Element_2\textbf{)}]\!] = S$

Input/Output
    Let $A_1 = relabel(ia[\![Element_1]\!])$
    Let $A_2 = relabel(ia[\![Element_2]\!])$
    Let $S = \emptyset$

Algorithm
    if in(startState$_{A_1}$) $\neq \emptyset$) then
        A$_1$ $\leftarrow$ addPreSpacing(A$_1$, startState$_{A_1}$)
    end if
    if in(startState$_{A_2}$) $\neq \emptyset$) then
        A$_2$ $\leftarrow$ addPreSpacing(A$_2$, startState$_{A_2}$)
    end if
    $S$ $\leftarrow$ addAll(A$_1$)
    $S$ $\leftarrow$ addAll(A$_2$)
    < S, startState$_S$ > $\leftarrow$ mergeStates(S, startState$_{A_1}$, startState$_{A_2}$)
    S $\leftarrow$ mergeEnds(S)
    for each endState in S do
        if out(S, endState) $\neq \emptyset$ then
            S $\leftarrow$ addPostSpacing(S, endState)
        end if
    end for
```

As for the definition of the send action above, we again give an example of how a decision in the UML4SOA concrete syntax maps into an IA. Figure 5.3 shows a simple decision in UML4SOA without any compensation handlers attached. To the left and right of the UML4SOA syntax, the individual IAs created for the left and right of the decision are shown.

The bottom of the figure shows the combined IA after the decision has been

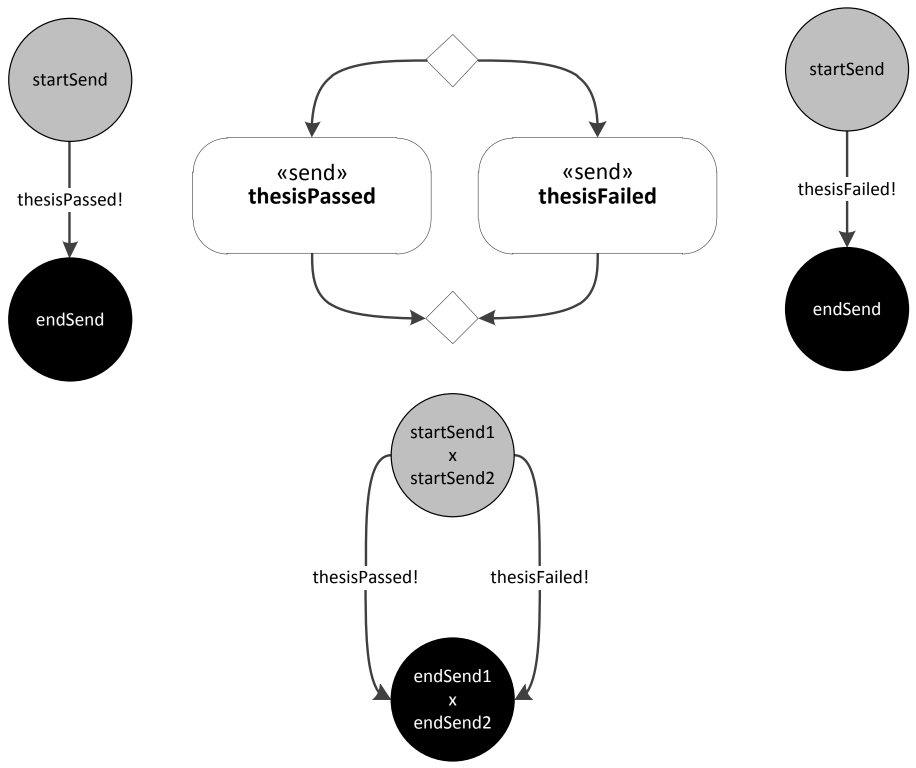


Figure 5.3: Example: A Decision in UML4SOA and as an IA

handled. Note that both automata have been relabelled — in this example, by suffixing the states with numbers, although this might also be achieved by some other means. As there are no compensation handlers, the decision algorithm was able to merge both ends into one; furthermore, as there are no loops, no spacing was required.

Parallel

Notation

parallel($Element_1$ | $Element_2$**)**

Intuition

Parallel execution of automata means that although each automata follows its own sequence of actions, these actions may occur in an arbitrarily interleaved sequence. The automata only reach an end if both have completed their own sequence.

A commonly used technique for achieving such a semantics is full interleaving; however, the fact that our automata distinguish between error- and non-error ends would require an additional clean-up after calculating the product, i.e. removing all edges and states after a combined state which includes an error state. Therefore, we interleave by traversing the automata from the combined start state.

Listing 5.14: MIO: Parallel

```
Definition
    $ia[\![\textbf{parallel(}Element_1; Element_2\textbf{)}]\!] = S$

Input/Output
    Let $A_1 = relabel(ia[\![Element_1]\!])$
    Let $A_2 = relabel(ia[\![Element_2]\!])$
    Let $S = \emptyset$

Algorithm
    S $\leftarrow$ $interleave(A_1,\, A_2,\, false)$
```

The interleaving process itself is performed by a helper function $interleave : IA \times IA \times Boolean \rightarrow IA$, which takes as parameters two intermediate automata and a boolean flag indicating whether the second automaton is to be interpreted as optional. In this case, the interleaving process allows the combined automaton to complete without the second. The interleaving algorithm is shown in listing 5.15.

Loop

Notation

loop(*Element***)**

Intuition

The intuition behind the loop function is to enclose an existing modal I/O automaton in a loop to enable to run it multiple times — at least once. The main challenge here is to ensure that if the loop contains error ends or multiple non-error ends, these are still available after a loop. This means:

1. If the original automaton ends with an error state, the whole loop must be aborted and the error state kept as an end error state of the loop.

2. If the original automaton ends with a non-error end state, the information attached to this state (installation of compensation handlers) must be kept as an end state of the loop. For example, consider an automaton with two endings — one in which a compensation handler A is installed, one in which a compensation handler B is installed. If we loop this automaton,

Listing 5.15: Handling Interleaving

```
Definition
    $interleave : IA \times IA \times Boolean \rightarrow IA$, $(A_1, A_2, opt) \mapsto S$

Algorithm
    $comps_S \leftarrow_{add} comps_{A_1}, comps_{A_2}$
    $<S, startState_S> \leftarrow newState(S)$
    $orig_{A_1} \leftarrow orig_{A_1}[\ startState_S \mapsto startState_{A_1}]$
    $orig_{A_2} \leftarrow orig_{A_2}[\ startState_S \mapsto startState_{A_2}]$
    $queue \leftarrow_{add} startState_S$
    $orig_{A_1} = [], orig_{A_1} = []$
    $seen \leftarrow \emptyset$
    while $queue \neq \emptyset$ do
        $state \leftarrow pop(queue)$
        $seen \leftarrow_{add} state$
        if $isErrEnd(A_1, orig_{A_1}(state))$ or $isErrEnd(A_2, orig_{A_2}(state))$ then
            $eeStates_S \leftarrow_{add} state$
        else if $isNormEnd(orig_{A_1}(state))$ and
                $isNormEnd(orig_{A_2}(state))$ then
            $neStates_S \leftarrow_{add} state$
        else
            if $opt$ and $isNormEnd(orig_{A_1}(state))$ and
                    $isStart(orig_{A_2}(state))$ then
                $neStates_S \leftarrow_{add} state$
            end if
            for each transition in $out(A_1, orig_{A_1}(state))$ do
                $< S, state_{combined} > \leftarrow newState(S)$
                $orig_{A_1} \leftarrow orig_{A_1}[\ state_{combined} \mapsto target(transition)]$
                $orig_{A_2} \leftarrow orig_{A_2}[\ state_{combined} \mapsto orig_{A_2}(state)]$
                $transitions_S \leftarrow_{add} (state, label(\underline{transition}), state_{combined})$
                if $state_{combined} \notin seen$ then
                    $push(queue, state_{combined})$
                end if
            end for
            for each transition in $out(A_2, orig_{A_2}(state))$ do
                $< S, state_{combined} > \leftarrow newState(S)$
                $orig_{A_1} \leftarrow orig_{A_1}[\ state_{combined} \mapsto orig_{A_1}(state))]$
                $orig_{A_2} \leftarrow orig_{A_2}[\ state_{combined} \mapsto target(transition)\ ]$
                $transitions_S \leftarrow_{add} (state, label(\underline{transition}), state_{combined})$
                if $state_{combined} \notin seen$ then
                    $queue \leftarrow_{add} state_{combined}$
                end if
            end for
        end if
    end while
```

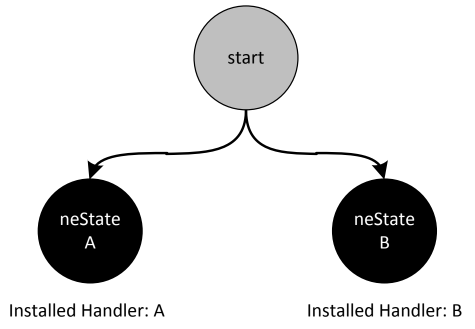


Figure 5.4: Semantics: An Automaton with Two Non-Error Ends

there are three possible ends: A installed, B installed, or both A and B installed.

Encountering an error end when dealing with loops effectively aborts the loop. Thus, in such cases, this end is kept as a non-error end and the path is not pursued further.

For non-error ends, we need to encode the information about which compensation handlers were installed in a loop run within the automaton itself. If the initial loop body — i.e., the first one the automaton runs into — ends with a non-error end state in which a certain set of compensation handlers S is installed, we need to keep this information, possibly adding more compensation handlers, until the loop is left.

We can do this by creating a copy of the original loop body for each possible set of installed compensation handlers after the loop. We say that this copy *corresponds* to the set of installed compensation handlers — i.e., whenever we handle a state in this copy, we have at least the compensation handlers installed which the copy stands for — plus possibly more when we reach a non-error end.

Thus, whenever we encounter a non-error end, we can either leave the loop or stay in the loop. In the latter case, we don't move back to the beginning of our current copy, but rather to the start of the copy which corresponds to the set of compensation handlers which includes a) the set of the current copy, and b) the set of the current non-error state.

The looping algorithm thus loops a child automaton while keeping the information about installed compensation handlers. There are two basic cases: If there are no non-error ends, a loop is not required. Otherwise, the automaton needs to be looped. An example of an automaton with two non-error ends with different compensation handlers installed is shown in figure 5.4.

The non-error end states of an automaton each correspond to a set of compensation handlers which are installed in the course of the automaton and are now available for later compensate calls. As the automaton may reach different end states in different loop runs, the installed compensation handlers at the end of the loop may include some or all of these sets. In particular, once the automaton has successfully completed by traversing to an end state, the set of compensation handlers installed in this end state must be stored until the loop is finished.

To achieve this, we multiply, or copy, the loop n times, where n is the size of the powerset of the installed compensation handler sets of the end states of the original automaton. This includes the empty set — corresponding to which compensation handlers are installed at the beginning of the loop — and every combination of possible sets of compensation handlers. The loop begins with the copy corresponding to the empty set. Once an end state is reached, the loop may be exited, or the automaton can continue in the copy which corresponds to the set of installed handlers in the end state.

An example of the automaton in figure 5.4 after loop handling is shown in figure 5.5.

Note that we add a new start state to the start of the loop. This is required as otherwise two consecutive loops can be executed out of order (i.e. it would be possible to execute the first loop again after the second). Furthermore, note that due to a loop, a compensation handler might get installed multiple times. In order to keep the resulting transition system finite, and in line with our disregard for data, we only note the fact that compensation was installed. Note that the handler might be executed multiple times as well (see section 5.1.1.5).

Although the algorithm (as usual) is directly defined on the syntax as input, we can also imagine a function $loop : IA \to IA$ implementing the body. We will use this function in later algorithms.

The algorithm for loops is shown in listing 5.16.

5.1.1.5 Activities

Notation

$$\textbf{activity(}\mathit{ActivityIdent}\ \textbf{:}\ \mathit{Element}\ \mathit{ActivityInterruptions}_{opt}\textbf{)}$$
$$\mathit{ActivityExceptions}_{opt}\ \mathit{ActivityEvents}_{opt}\ \mathit{ActivityCompensation}_{opt}$$

Intuition

An activity is the main structured concept of the SMM. An activity includes children and has attached handlers. We have already defined what a child can look like; mostly, it will be a structured element. An activity has a name, children, several (including zero) exception handlers with each specify which exception they handle, several (including zero) event handlers, and zero or one compensation handler. Furthermore, it may have interrupting receives which can be invoked at any time, ending the activity.

The intuition for handling an activity is more complex than for any other element. The following list gives an overview of the process of handling an activity; they will be detailed in the following sections.

1. First of all, an activity is a grouping concept, therefore it has a *child* which must be executed as part of the activity.

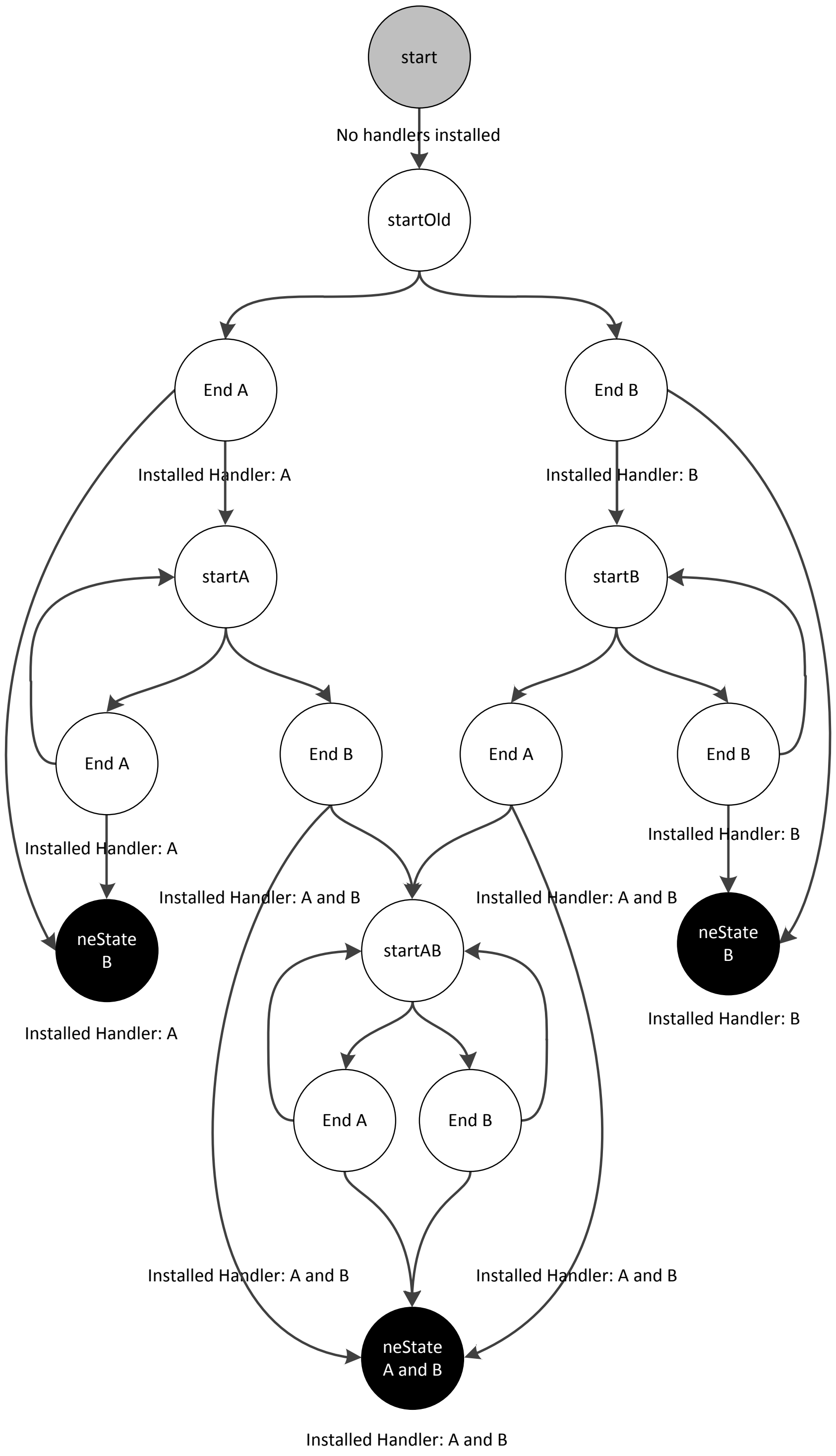


Figure 5.5: Semantics: Looping with Compensation Handlers

Listing 5.16: MIO: Loops

```
Definition
    $ia[\![\ \textbf{loop(}Element\textbf{)}\ ]\!] = S$

Input/Output
    Let $A = relabel(ia[\![Element]\!])$
    Let $S = \emptyset$

Algorithm
    if $neStates_A = \emptyset$ then
        S ← A
    else
        cSet ← $\wp$(cInstallTransitions(A, endState$_{A,1}$) ...
                cInstallTransitions(A, endState$_{A,n}$))
        automataSet ← $\emptyset$
        for each installSet in cSet ordered by size descending do
            cur ← relabel(A)
            cur ← addPreSpace(cur)
            cur ← addPostSpace(cur)
            automataSet $\leftarrow_{add}$ (installSet ↦ cur)
            S ← addAll(states$_{cur}$, <$\emptyset$, eeStates$_{cur}$>, transitions$_{cur}$,
                    comps$_{cur}$)
            if $installSet = \emptyset$ then
                startState$_S$ ← startState$_{cur}$
            end if
            for each nonErrorEnd in neStates$_{cur}$ do
                combinedSet ← installSet ∪
                        cInstallTransitions(S, nonErrorEnd)
                target ← automataSet(combinedSet)
                transitions$_S$ $\leftarrow_{add}$ (nonErrorEnd, $\underline{\text{loop;}}$, startState$_{target}$)
            end for
        end for
    end if
```

2. If *interrupting receives* are specified, these must be possible at any time during the execution of the child element.

3. An activity may have *event handlers*, which is additional behaviour which may run, at any time, zero, once, or multiple times, in parallel to the activity. Therefore, this behaviour needs to be *weaved* into the normal behaviour given by the children.

4. An activity may have *exception handlers*, which specify behaviour to be executed if a certain exception (attached to the handler) is thrown in the activity, i.e. within the behaviour of the child.

5. An activity may have a *compensation handler*. If this is the case, the activity may later be *undone* with the behaviour in the handler, thus we need to store the handler for later use. Also, we need to make sure that for each outgoing non-error state from this activity, it is known that we have successfully installed a compensation handler.

6. Finally, the exception- or compensation handlers of the activity may have *compensate calls*, i.e. invocation of formerly installed compensation handlers. We therefore need to check such calls to see if we have installed compensation handlers we can execute at the site of the compensate call within the handler.

Due to the complexity of activity handling, we split the definition of the algorithm into these six steps and define a new function for handling each step. See listing 5.17 for the activity-handling algorithm.

- The first function to execute is $handleInterruption : IA \times IA \rightarrow IA$. It merges the second parameter (the interruption) with the first parameter (an activity automaton).

- The next function to execute is $handleException : IA \times String \times IA \rightarrow IA$. This function checks whether the automaton given as the first parameter contains any dangling error ends matching the exception identifer given as the second parameter; if so, the automaton given as the third parameter is added as the exception handler.

- For all event handlers, $handleEvent : IA \times IA \rightarrow IA$ is called, which weaves the event handler Ev into the automaton S as appropriate.

- The function $handleCompensationCalls : IA \rightarrow IA$ looks for compensation invocations inside the given automaton, replacing them with logic from already installed compensation handlers.

- Finally, $handleCompensation : IA \times IA \rightarrow IA$ ensures that the compensation handler given as the second parameter is added to the main automaton given as the first parameter.

In the following, we define the functions used in the above definition.

Listing 5.17: MIO: Activities

```
Definition
    $ia[\![\textbf{activity(}sIdent : Element_{Ch}\ \textbf{interrupt}\ Receive_1 \ldots$
            $\textbf{interrupt}\ Receive_m\textbf{)}$
      $\textbf{exception}\ ident_{X_1}\colon\ Exception_{X_1}\ \ldots ident_{X_n}\colon\ Exception_{X_n}$
      $\textbf{event}\ Element_{e_1}\ \ldots\ \textbf{event}\ Element_{e_o}$
      $\textbf{compensation}\ Element_{co}\ ]\!] = S$

Input/Output
    Let $A = relabel(ia[\![Element_{Ch}]\!])$
    Let $R_1 \ldots R_m = relabel(ia[\![Receive_1]\!]) \ldots relabel([\![Receive_m]\!])$
    Let $X_1 \ldots X_n = relabel(ia[\![Element_{x_1}]\!]) \ldots relabel([\![Element_{x_n}]\!])$
    Let $Ev_1 \ldots Ev_o = relabel(ia[\![Element_{e_1}]\!]) \ldots relabel([\![Element_{e_o}]\!])$
    Let $C = relabel(ia[\![Element_{co}]\!])$
    Let $S = \emptyset$

Algorithm
    S ← addAll(A)
    evAttStates$_S$ ← states$_S$

    for each R in R$_1$ ...R$_n$ do                              ▷ Interruptions
        S ← handleInterruption(S, R)
    end for

    for each X in X$_1$ ...X$_n$ do                          ▷ Exception Handling
        S ← handleException(S, ident$_X$, X)
    end for

    for each Ev in Ev$_1$ ...Ev$_m$ do                          ▷ Event Handling
        S ← handleEvent(S, Ev)
    end for

    S ← handleCompensationCalls(S)                        ▷ Compensation Calls
    S ← handleCompensation(S, sIdent, C)                ▷ Compensation Handler
```

Interruptions

The algorithm defined in function *handleInterruption* adds a receiving transition to each of the states in the child automaton which is to be interrupted, except for error ends. Each receiving transition leads to a new end state, as the child automata may have different compensation handler installations.

Newly created end states which are equal with regard to compensation installations are merged at the end.

Listing 5.18: Handling Interruptions

```
Definition
    $handleInterruption : IA \times IA \to IA$, (A, R) $\mapsto$ S

Algorithm
    S $\leftarrow$ A
    neStates$_S$ $\leftarrow \emptyset$
    for each state in states$_S$ do
        if not state $\in$ errStates$_S$ then
            R$_{copy}$ $\leftarrow$relabel(R)
            S $\leftarrow$ addAll(R$_{copy}$)
            < S , _ > $\leftarrow$ mergeStates(S, state, startState$_{R_{copy}}$)
        end if
    end for
    S $\leftarrow$ mergeEnds(S)
```

Exception Handlers

SMM activities may throw exceptions which may be handled by an exception handler attached to a service activity. In an IA, an exception is encoded as an *error end state*, indicating that the normal flow of execution ends at this point. An exception handler can be used to handle the exception, and thus allow subsequent behaviour after the error end.

Adding exception handling is only possible if we have matching exception handlers installed — i.e., if the exception corresponding to a handler is the same one as the exception corresponding to an error end state.

The algorithm for handling exceptions is given in listing 5.19. An example for exception handling is shown in figure 5.6. Three IAs are shown. The first (left) is the existing automata A in which exceptions are to be handled. The second (middle) is the exception handling IA itself, which is associated in the textual syntax with the same exception *myFault* thrown in the IA on the left. Finally, the right-hand side shows the resulting automaton S. Note that the former error end state in the middle is now a normal state and has been merged with the start state of the error handling automaton. This error end will thus not be handled again. The final state is a non-error state.

Listing 5.19: Handling Exceptions

```
Definition
    $handleException : IA \times String \times IA \rightarrow IA$, (A, ident, X) $\mapsto$ S

Algorithm
    S $\leftarrow$ A
    for each errEnd in eeStates$_S$ do
        if exceptionsMatch(in(errEnd), ident) then
            S $\leftarrow$ addAll(X)
            < S , _ > $\leftarrow$ mergeStates(S, errEnd, startState$_X$)
            eeEnds$_S$ $\leftarrow$ eeEnds$_S$ $\setminus$ errEnd
        end if
    end for
```

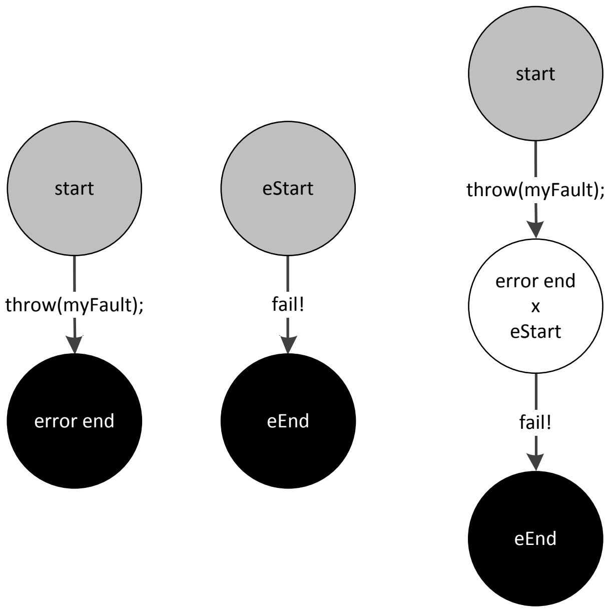


Figure 5.6: Example: Interruptions

Event Handlers

An event handler must run in parallel to the normal children of the activity — it may be executed multiple times, but also not at all. Therefore, we can reuse the algorithms already defined for looping and parallel behaviour; the only difference is that an event handler is *optional*.

Thus, the algorithm for event handling is quite short:

Listing 5.20: Handling Events

```
Definition
    $handleEvent : IA \times IA \to IA$, (A, Ev) $\mapsto$ S

Algorithm
    S $\leftarrow$ $interleave$(A, $loop$(Ev), $true$)
```

Compensation Calls

A *compensate*(*activity*) or *compensateAll* transition is added to an automaton during the handling of an exception- or compensation handler. Now that the handler is added to a scope, compensation handlers are available which might need to be handled. By traversing the automaton back to the start state, a list of installed compensation handlers can be identified. Note that ordering is important in the case of *compensateAll*, as we want to compensate activities in the reverse order of execution.

Actual compensation takes place by replacing the *compensate*(*activity*) or *compensateAll* transitions with an (inverse) concatenation of compensation handlers.

See listing 5.21 for the compensation call algorithm.

Compensation Handler

A compensation handler is behaviour which rolls back an activity. The behaviour can be used later in the automaton in a compensate call.

Listing 5.22: Handling Compensation Handlers

```
Definition
    $handleCompensation : IA \times String \times IA \to IA$, (S, name, C) $\mapsto$ S

Algorithm
    comps$_S$ $\leftarrow_{add}$ $name \mapsto$ C
    for each nonErrorEndState in neStates$_S$ do
        <S, newEndState> $\leftarrow$ newState(S)
        neStates$_S$ $\leftarrow$ neStates$_S$ $\cup$ newEndState $\setminus$ nonErrorEndState
        transitions$_S$ $\leftarrow_{add}$ (nonErrorEndState,
                $\underline{\text{compInstalled(name)}}$, newEndState)
    end for
```

We need to do two things on the spot:

- We need to add the information that a compensation handler exists for the current activity. This effectively means adding the compensation handler to *comps* of the main automaton.

Listing 5.21: Handling Compensation Calls

```
Definition
    $handleCompensationCalls : IA \rightarrow IA$, S ↦ S

Algorithm
    for each call in cCallTransitions(S) do
        installTransitions ← cInstallTransitions(S, call)
        assignTo ← source(call)
        for each iTransition in installTransitions do
            if compensationsMatch(call, iTransition) then
                C ← comps($actName$(iTransition))
                < C , end > ← mergeStates(C, neStates$_C$)
                if isInLoop(S, iTransition) then
                    transitions$_C$ $\leftarrow_{add}$ (end, $\underline{\text{loop;}}$, startState$_C$)
                    C ← addPreSpacing(C, start$_C$)
                end if
                S ← addAll(C)
                < S , _ > ← mergeStates(S, assignTo, startState$_C$)
                assignTo ← end
            end if
        end for
        <S , handledState > $\leftarrow_{add}$ newState(S)
        transitions$_S$ $\leftarrow_{add}$ (assignTo, $\underline{\text{compHandled}(actName\text{(iTransition)})}$,
            handledState)
        < S , _ > ← mergeStates(S, target(call), handledState)
        transitions$_S$ ← transitions$_S$ \ call
    end for
```

- Furthermore, we need to add the information about the installed handler to the main activity automaton. This is done by adding an internal transition with the label *compInstalled(name)*.

The definition of $handleCompensation$ concludes the discussion of activity conversion. Thus, all functions for converting SMM behavioural service descriptions have been introduced.

5.1.2 SMM Protocols

An SMM protocol is specified as a state machine; as discussed in chapter 3 and further detailed in chapter 6, it maps to a UML protocol state machine in UML4SOA.

Our semantic domain for the SMM, modal input/output transition systems, is already very close to the SMM definition of a protocol, which consists of the following elements:

- *States.* The set of states; one state is denoted as the start state.
- *Transitions.* A transition is a directed link between two states.
  - A transition is either a `SendingTransition`, a `ReceivingTransition`, a `ReplyingTransition`, a `ReceiveReplyingTransition`, or a `NoopTransition`.
  - A transition may furthermore be *optional* if the `isOptional` flag is set.
  - A transition may be linked to an `InterfaceOperation` by means of the `operation` association.

Recalling from chapter 2, a modal input/output transition system consists of the following elements:

- *States.* One state is distinguished as a start state; the others are a flat set.
- *Input, Output, and Internal Actions.* Input actions correspond to received messages, output actions to sent messages, and internal actions to non-external steps.
- *May and Must Transitions.* Transitions are directed edges between two states; they are either must or may.

From the two above lists, it follows that the mapping is rather straightforward. We can give a simple algorithm which achieves this by iterating over all input elements:

- Each state in the SMM protocol is converted to a MIO state. The resulting converted state of the start state is set as the start state of the MIO.

- Each operation linked from a transition of the SMM protocol is converted to a MIO action as follows:
  - For `SendingTransition`s, the operation name of the linked `operation` is used as the label of an *output* action.
  - For `ReplyingTransition`s, the operation name of the linked `operation` is used as the label of an *output* action, but is prefixed with "return_".
  - For `ReceivingTransition`s, the operation name of the linked `operation` is used as the label of an *input* action.
  - For `ReceiveReplyingTransition`s, the operation name of the linked `operation` is used as the label of an *input* action, but is again prefixed with "return_".
- Each transition of the SMM protocol is converted into a MIO transition. Depending the state of the `isOptional` flag of the SMM transition, the corresponding set is chosen. The label of the transition depends on the transition type: For `SendingTransition` and `ReplyingTransition`, the corresponding output label for the operation of the transition is selected; for `ReceivingTransition` and `ReceiveReplyingTransition`, the corresponding input label. In case of a `NoopTransition`, we use *tau* from the internal action set.

This simple algorithm suffices for the conversion of SMM protocols to modal input/output transition systems.

5.1.3 A Complete Example

While the previous sections have discussed the translation of SMM models into MIOs for each individual construct of services and protocols, the case study we have already introduced in chapter 3 uses (nearly) all of these elements in combination. In the following, we thus give an example of how the case study (modelled in UML4SOA) is translated into MIOs. As UML4SOA is the concrete syntax for the SMM, we skip displaying the SMM meta-model.

Recalling from chapter 3, the central participant of the case study — the *ThesisManager* process — is shown in figure 5.7. Mapping this process to MIOs yields the transition system shown in figure 5.8. Comparing these two figures to one other yields several interesting insights.

First, the obvious fact to note is the conversion of the basic communication actions into transitions in the MIO. For example, the first action in the UML-4SOA activity — the *createThesis* receive — appears as a transition with the label *createThesis* and the suffix *?* indicating that the label originates from within the input action set in the MIO. Likewise, each send operation in the UML4SOA model appears as a label from the output action set (indicated by the *!* suffix).

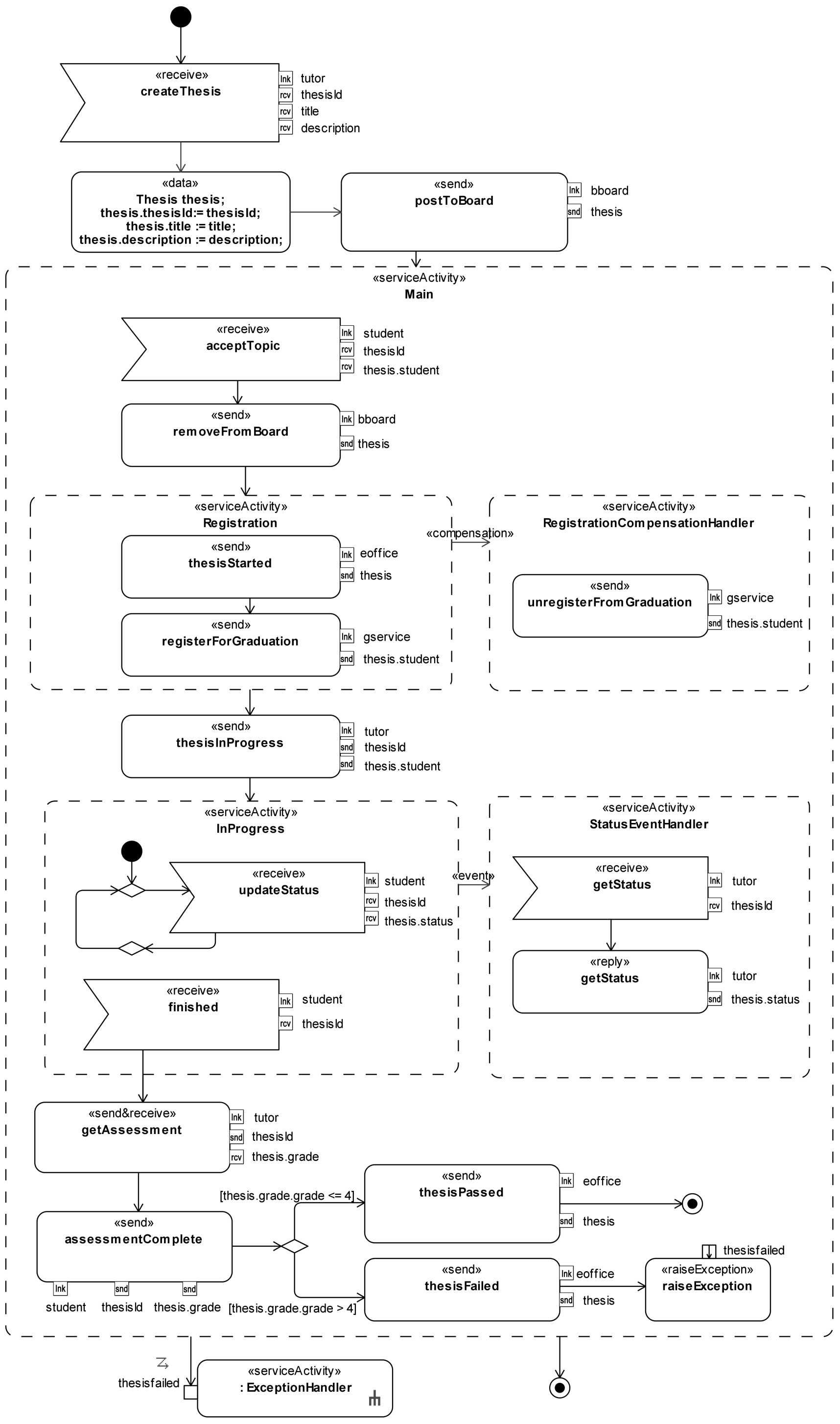


Figure 5.7: Thesis Manager in UML4SOA

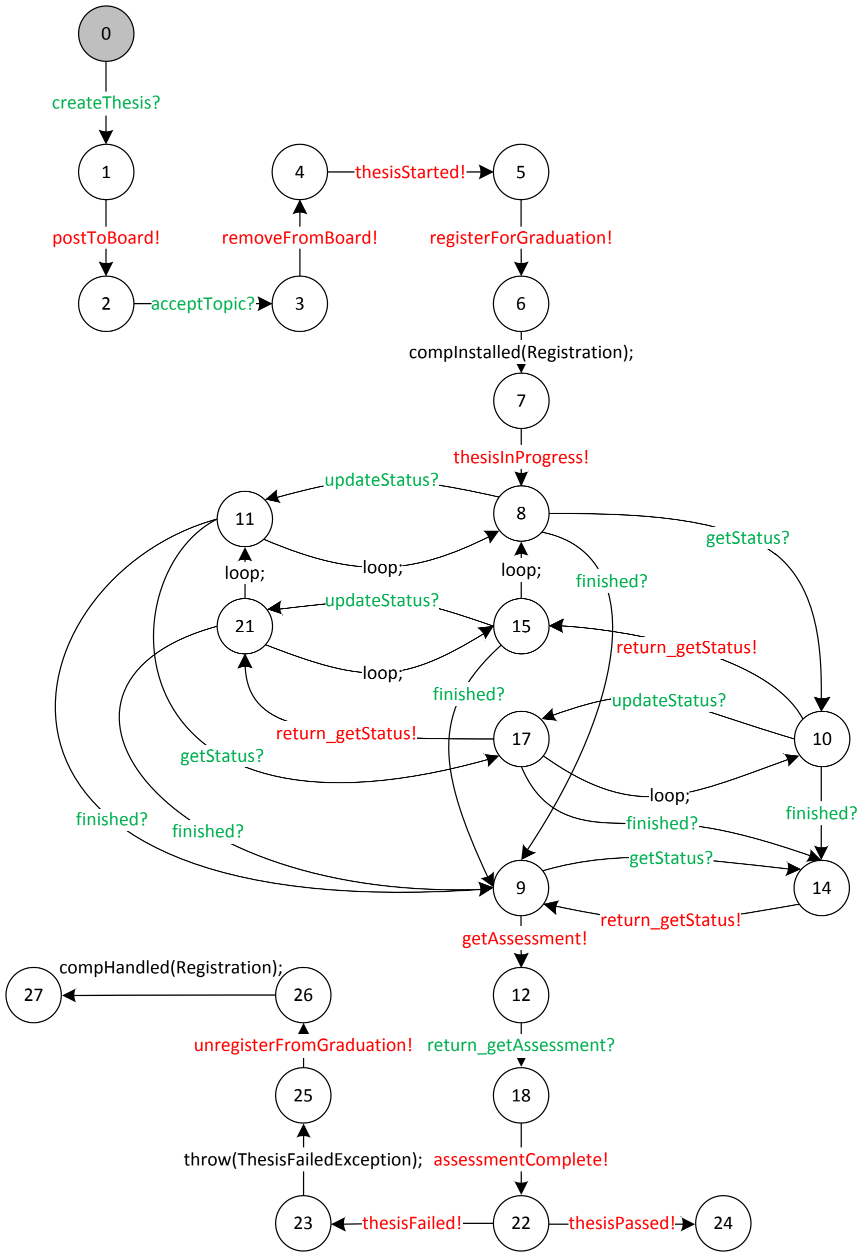


Figure 5.8: Thesis Manager as a MIO

Second, after the thesis is in progress, we enter a loop in which the student may provide updates. The loop occurs multiple times as it is interleaved with the event handler and the interrupting action *finished* — at various places, the *updateStatus* and corresponding internal *loop* action can be seen.

Third, note how events are handled in the MIO: The loop which contains the *updateThesis* call has an event handler — thus, in each state which is part of the loop in the MIO, the *getStatus* call can be received (and replied to, even if there are interleaved calls in-between).

As an automaton does not contain specific support for compensation, the (single) compensation handler of the process is added at the appropriate invocation site. In this case, the invocation of the compensation handler is executed as part of the exception handler of the *Registration* scope, where *compensate* is invoked. Thus, after the *throw* actions in the automaton, the actions of the compensation handler are inserted.

Finally, note that the *finished* receive in the UML4SOA process with its interrupting edge effectively aborts the *InProgress* scope. Thus, from each state which is part of the inner loop, a *finished* transition leads to the final part of the process which handles cleaning up and reporting the state of the thesis.

Converting UML4SOA protocols is easier and more straightforward. An example for a protocol conversion is shown in figure 5.9: On the left hand side, the *Student* protocol in UML4SOA is shown; on the right-hand side, the corresponding MIO is displayed.

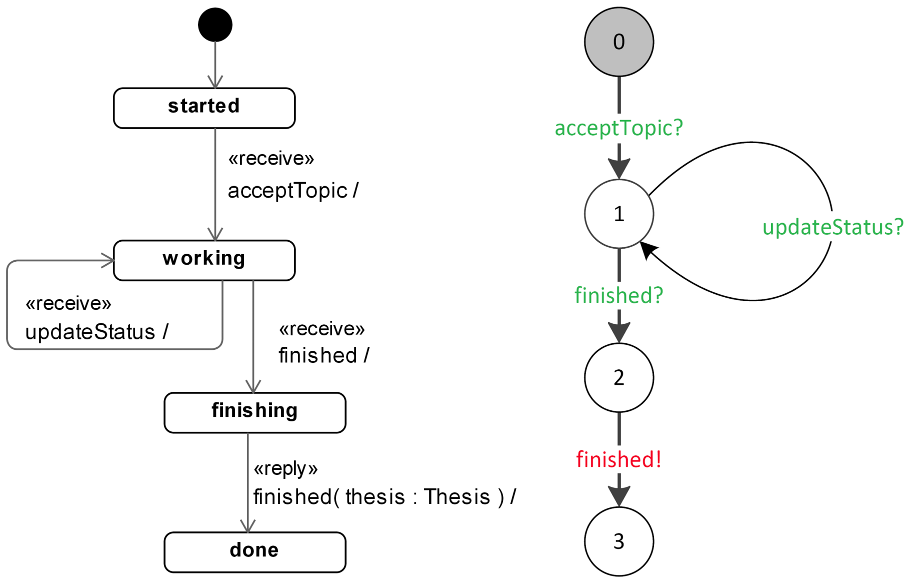


Figure 5.9: Student Protocol: UML4SOA (left), MIO (right)

5.1.4 Conclusion

This section has introduced a semantics for both SMM participant behaviour and SMM service protocols. Firstly, we have defined the function $ia[\![]\!]$ which transforms the textual syntax of SMM participant behaviour to modal I/O automata via intermediate automatas (IAs). As the UML, SoaML, and the UML4SOA/Strict profile form the concrete syntax of the SMM, the semantics can directly be used for UML4SOA models. Secondly, we have introduced a direct mapping from SMM service protocols to MIOs, and thus, also for SOA participant protocols modelled in UML4SOA/Strict (which are specialised UML PrSMs).

Both mappings from SMM participant behaviours and service protocols to MIOs are defined through deterministic algorithms with guaranteed termination; they are hence well-defined. Termination of the given algorithms can be shown as follows: All inputs of the behaviour semantics function are based on the well-nested SMM structure; they do not contain any links back to potentially already visited nodes which might lead to infinite behaviour. Of the functions with IAs as input, only one traverses the input graph: The *interleave* function uses a queue to walk through existing IAs. To avoid termination problems, the corresponding algorithm uses a cache to note already visited nodes (in the *seen* variable). Finally, the protocol semantics function does not change the structure of the data, directly converting states to states and transitions to transitions; it thus also does not suffer from termination problems.

As we show in later sections, the pseudo-code given above has been fully implemented in a tool and used to transform several case studies.

With the above definitions, we have defined a rigorous semantics for SMM participants and protocols which can be used for formal analysis, checking, validation, and verification of UML4SOA models.

5.2 Analysing UML4SOA Models

A formal semantics such as the one introduced for UML4SOA in the previous sections forms the basis for rigorous analysis and verification of the modelled SOA system. The semantic domain used in our case — modal input/output transition systems — is amenable to various kinds of analyses. In chapter 2, we have introduced the notion of *interface theories* for precisely specifying and answering two very important questions to be asked of a system model which are directly relevant for the developers. In the context of services and service protocols, these questions can be reformulated as follows:

1. Does a service correctly implement a given protocol?

2. Do two given protocols work together correctly?

Again, it is important to note that these two questions are directly related to one another: Usually, given two protocols which can work together and a participant behaviour which implements one of these, we want to be able to assume that the participant can work with the other protocol as well. Interface theories formalise this idea, precisely defining the two questions given above as *refinement* and *compatibility*, and giving the assurance that *compatibility is ensured under refinement*. A graphical overview of how the intuitive notion of the questions given above map to the formal specifications is given in figure 5.10.

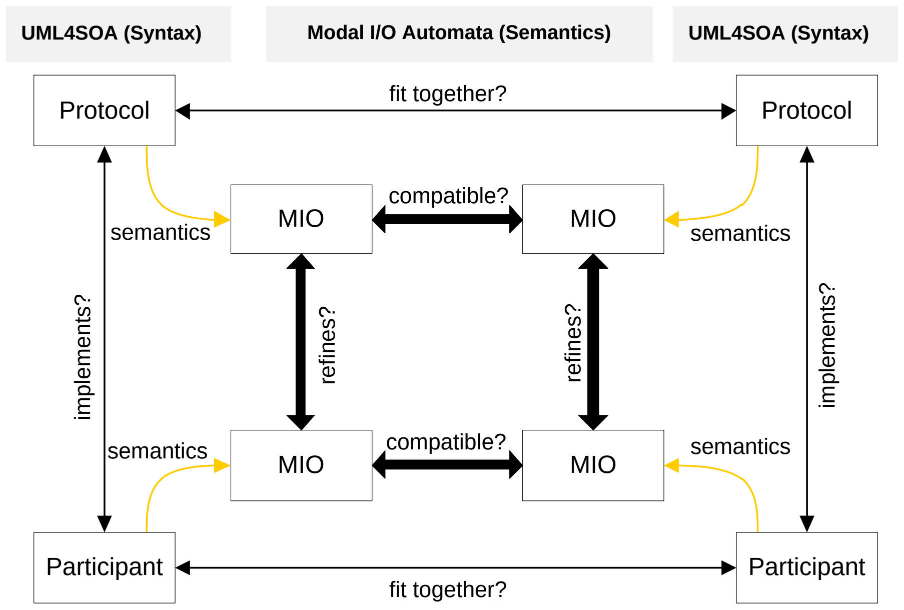


Figure 5.10: UML4SOA Analysis: Syntax & Semantics

Various interface theories and, along with them, various notions of refinement and compatibility have been defined in the literature; each having a different application area and thus differing in the details of which MIOs are regarded as refinements or as compatible.

We first discuss analysis with *weak modal refinement and compatibility* as introduced in chapter 2, which is focused on race conditions. Afterwards, we introduce an additional interface theory which allows a protocol-based analysis focused on deadlocks not based on race conditions.

5.2.1 Analysis with Weak Refinement and Compatibility

Chapter 2 has introduced the *weak* interface theory. This theory is useful in the process of checking the refinement of a service to a service protocol, and in checking protocol compatibility as it enables bypassing internalised actions without sacrificing rigorous verification with regard to race conditions.

A service is checked against its protocols in a pair-wise fashion, i.e. separately for each protocol. For checking participant behaviours with more than

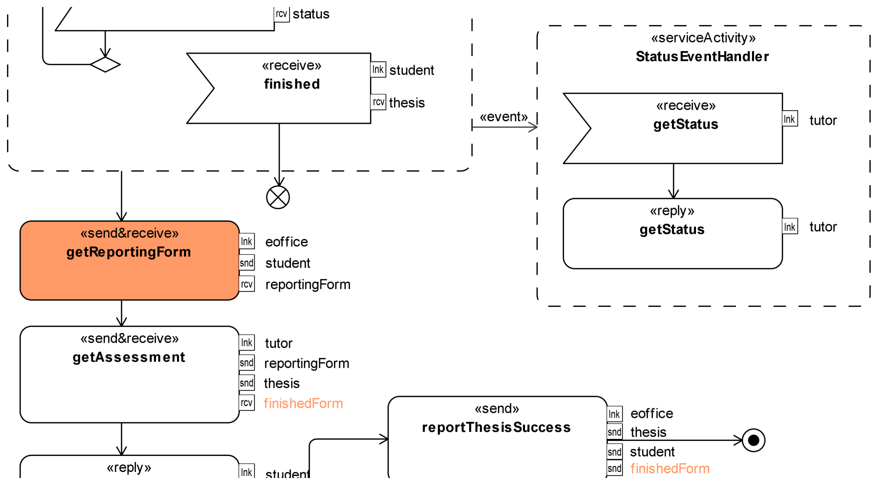


Figure 5.11: Incorrect Excerpt from Case Study (1)

one partner, we additionally need the notion of *hiding* as introduced in chapter 2 to internalise actions not present in the current protocol to be checked.

As an example for using *weak* refinement and compatibility, we consider again our case study. The `ThesisManagement` process and its MIO representation illustrated on pages 158 and 159 can be shown to be a refinement of all protocols (which are shown on page 160). To illustrate the analysis, we will therefore introduce an error into the participant behaviour.

Consider the excerpt from the case study shown in figure 5.11. In this example, the tutor is required to fill out a personalised form for assessing the student, which is to be retrieved from the examination office before the assessment can take place. A new action has been added for retrieving the form, and the subsequent interaction with the tutor and examination office uses the form. Furthermore, the call has also been added to the expected protocol from the examination office (not shown).

Running a weak refinement analysis on the MIO representation of the `Tutor` protocol and the `ThesisManager` participant reports a race condition between the `getStatus` receive and the `getAssessment` send&receive: During the `getReportingForm` call, the tutor does not yet know that the thesis is finished and thus, nothing restricts him from sending out another `getStatus` call which cannot be accepted anymore by the participant behaviour.

Race conditions like the one shown here are in fact common and are reliably found by checking for weak refinement. In our case, this problem can be alleviated by first informing the tutor about the finished state of the thesis (before `getReportingForm`).

In the next section, we revisit this example, showing what other types of errors can occur and how to detect them.

5.2.2 Strict-Observational Analysis

A key application area of UML4SOA is designing SOA models early in the development process, where the focus is on the *big picture* of the resulting system. In this context, we believe that is important to support developers in answering the two questions discussed in the beginning of this section on a higher level than it is currently possible.

Weak modal refinement and compatibility already impose very strict requirements on the system design; their results include checking for race conditions which may, but need not necessarily lead to problems during execution. While these checks are necessary in the end, they detract from more obvious and critical problems during the initial design phase, where the focus is on the overall design of the system.

Thus, we have chosen to create an additional interface theory which is focused on the externally visible behaviour of services. The difference between the two notions of weak and strict-observational checking lies in the treatment of *race conditions*. A race condition *may* (in one path) lead to a *deadlock*, but it is possible to use another path which succeeds. On the other hand, there are deadlocks which are not race conditions; these cannot be avoided when following the protocol from the start state. Our new notions of refinement and compatibility ignore deadlocks associated with race-conditions; they instead focus on deadlocks which will always be reached.

Thus, this form of analysis enables checking if a participant behaviour is able to follow the possible calls described in a protocol, or if two protocols can follow each other's call sequences. A positive answer guarantees that there is no hard deadlock, i.e. it is possible to follow the correct path in the behaviour. On the other hand, a negative answer indicates that either the service or the protocol must be corrected in order to work *at all*.

For introducing the formal notions of *strict-observational refinement and compatibility*, we use a simple example with a participant behaviour `O` and two protocols `P` and `R`, shown in figure 5.12. In the next section, we apply the new interface theory to the case study.

We first try to analyse this service with existing interface theories for modal I/O automata. The closest to our own definitions of refinement and compatibility for MIOs are weak refinement and compatibility, complemented by the notion of hiding.

5.2.2.1 Refinement

Let us first consider checking the participant behaviour `O`, focusing on the question of whether it implements (refines) the `P` protocol with *weak refinement*.

First, we need to hide actions not in `P`. Recalling from chapter 2, the notion of hiding allows moving an action label from the output or input set of the automaton to the internal set, i.e., effectively hiding it from the external alphabet. In our example, this applies to the action `e?` which is from a different protocol (`R`) than the one we are currently checking (`P`).

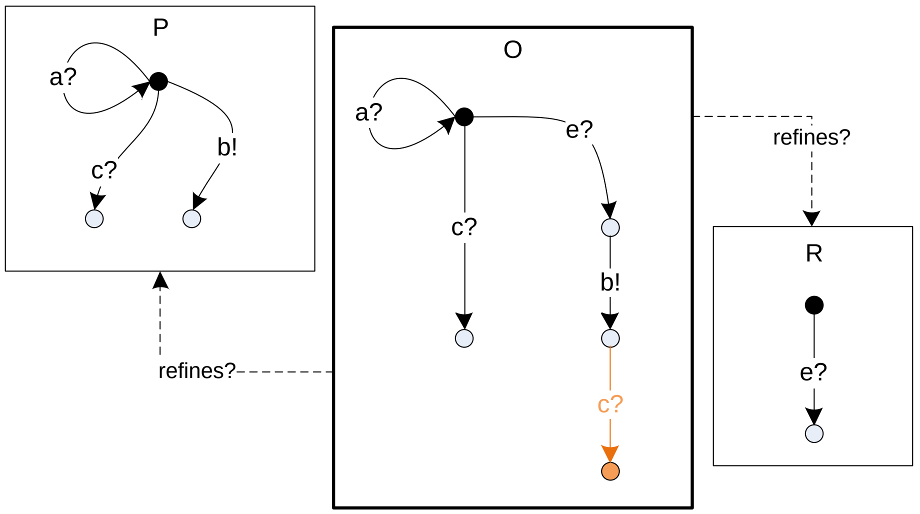


Figure 5.12: MIO Example for Strict-Observational Analysis (1)

Now, using weak refinement, we get the following problem report: After having taken the now-internal `e;` call, the protocol action `c?` is no longer possible. This is a classic race condition.

As indicated above, we are not interested in deadlocks originating in race conditions in our new interface theory. Rather, we want to report deadlocks which can not be bypassed in a race. Thus, focusing only on actions of P, we notice that the participant behaviour waits for another `c?` action (illustrated in red) after `b!`. This action is *never* allowed in `P` and the implementation will *always* deadlock at this point.

Strict-observational refinement is geared towards reporting such problems.In other words, instead of detecting errors that *may* lead to a deadlock in *one* path, strict-observational refinement reports deadlocks which occur in *all* relevant paths — i.e., situations in which the protocol *cannot* be followed by the participant behaviour. The other problem indicated above (`c?` not being possible after `e?`) is not detected by our refinement notion, but can be readily dealt with using weak refinement.

In the following, we formalise the notion of strict-observational refinement motivated above. Please refer back to chapter 2 for the basic definitions of MIOs, refinement, and transition types.

To capture the idea of strict-observational analysis, we first define the notion of *action-weak transitions.*

Given a MIO S, a set of actions $L \subseteq act_S$, and an action $a \in act_S$, we write $s \xrightarrow{a}_S^{\triangleleft(L)} s'$ iff either $s \xrightarrow{a}_S s'$ or there exist $n \geq 1$, states $s_1, \ldots, s_n \in states_S$, and actions $b_1, \ldots, b_n \in (act_S \setminus L)$ such that

$$s \xrightarrow{b_1}_S s_1 \ldots s_{n-1} \xrightarrow{b_n}_S s_n \xrightarrow{a}_S s'.$$

The intuition of an *action-weak transition* is that a single relevant action is performed, which may be preceded by irrelevant actions not in L. Moreover, we write $s \xrightarrow{a}{}^{\lhd}_S s'$ to abbreviate $s \xrightarrow{a}{}^{\lhd(ext_S)}_S s'$. The same notions are analogously used for may-transitions.

We now adapt weak modal refinement to satisfy our needs, i.e. we only want to consider relevant actions during refinement of MIOs. The basic idea for our refinement is to *skip* leading actions unrelated to the protocol under investigation. First, the refining MIOs may have more actions than the refined one, and second, in both directions in the definition we focus on the external actions of the more abstract MIO since these actions are the relevant ones.

Definition 10 (Strict-Observational Modal Refinement) *Let S and T be MIOs such that $\alpha_{ext}(S) \supseteq \alpha_{ext}(T)$. S strict-observationally refines T, denoted by $S \leq_{so} T$, iff there exists a relation $R \subseteq states_S \times states_T$ containing $(start_S, start_T)$ such that for all $(s,t) \in R$, for all actions $a \in ext_T$,*

1. *if $t \xrightarrow{a}{}^{\lhd}_T t'$ then there exists $s' \in states_S$ such that $s \xrightarrow{a}{}^{\lhd(ext_T)}_S s'$ and $(s', t') \in R$,*
2. *if $s \overset{a}{\dashrightarrow}{}^{\lhd(ext_T)}_S s'$ then there exists $t' \in states_T$ such that $t \overset{a}{\dashrightarrow}{}^{\lhd}_T t'$ and $(s', t') \in R$.*

Note that strict-observational refinement only uses the modal aspects of MIOs and can thus also be defined for modal transition systems in their original form (i.e. not distinguishing between input/output/internal actions).

Strict-observational modal refinement can be proved to be a preorder on MIOs, i.e. it is reflexive and transitive.

Lemma 1 *Strict-observational modal refinement $\leq_{so}$ is reflexive and transitive.*

Before we go on to prove this lemma, we include (and prove) the following helper lemma:

Lemma 2 *Let S be a MIO and $a \in act_S$. If $s \xrightarrow{a}{}^{\lhd(L)}_S s'$ and $L' \subseteq L$ then $s \xrightarrow{a}{}^{\lhd(L')}_S s'$.*

Proof 1 *of Lem. 2 Let $s \xrightarrow{a}{}^{\lhd(L)}_S s'$ and $L' \subseteq L$. By definition of $\longrightarrow^{\lhd(L)}$, we must distinguish two cases.*

1. *$s \xrightarrow{a}_S s'$. Here, $s \xrightarrow{a}{}^{\lhd(L')}_S s'$ holds.*
2. *there exist $n \geq 1$, states $s_1, \ldots, s_n \in states_S$, and actions $b_1, \ldots, b_n \in (act_S \setminus L)$ such that*

$$s \xrightarrow{b_1}_S s_1 \ldots s_{n-1} \xrightarrow{b_n}_S s_n \xrightarrow{a}_S s'.$$

Since $L' \subseteq L$, it holds for all b_i that $b_i \in (act_S \setminus L')$. From this we can deduce that $s \xrightarrow{a}_S^{\triangleleft(L')} s'$ holds.

Proof 2 *of Lem. 1 (Reflexivity and Transitivity of Refinement)*
As reflexivity of refinement is easy to show, we focus on showing transitivity.

Let $S \leq_{so} U$ and $U \leq_{so} T$, we have to show that $S \leq_{so} T$. We can assume a strict-observational modal refinement R_1 for S and U, and a strict-observational modal refinement R_2 for U and T. We now define a relation $R \subseteq states_S \times states_T$ by

$$R = \{(s,t) \mid \exists u \in states_U.(s,u) \in R_1 \wedge (u,t) \in R_2\}.$$

We show that R is a strict-observational modal refinement for S and T. First, it holds that $(start_S, start_T) \in R$. Now, we have to show (i) and (ii) from Definition 10 of strict-observational modal refinement.

1. *(Protocol to Implementation). Assume $(s,t) \in R$ and*

$$t \xrightarrow{a}_T^{\triangleleft} t'$$

for some $a \in ext_T$ (only consider externals as per definition of $\leq_{so}$). By definition of R, there exists $u \in states_U$ such that $(s,u) \in R_1$ and $(u,t) \in R_2$. From R_2 it follows that there exists a state $u' \in states_U$ such that

$$u \xrightarrow{a}_U^{\triangleleft(ext_T)} u'$$

and $(u',t') \in R_2$. It follows that we have

$$u \xrightarrow{b_1}_U^{\triangleleft} \ldots \xrightarrow{b_n}_U^{\triangleleft} \hat{u} \xrightarrow{a}_U^{\triangleleft} u'$$

for some $n \geq 0$ (if $n = 0$ then $u = \hat{u}$) and for some $b_i \in ext_U \setminus ext_T$, $1 \leq i \leq n$ (if $n > 0$). By assumption we know $(s,u) \in R_1$. By induction on the number of transitions between u and $\hat{u}$ in this trace we can show that there exist transitions

$$s \xrightarrow{b_1}_S^{\triangleleft(ext_U)} \ldots \xrightarrow{b_n}_S^{\triangleleft(ext_U)} \hat{s} \xrightarrow{a}_S^{\triangleleft(ext_U)} s'$$

such that $(s',u') \in R_1$. By Lemma 2 and because of $b_i \notin ext_T$ we get

$$s \xrightarrow{a}_S^{\triangleleft(ext_T)} s',$$

thus, by definition of R, we get $(s',t') \in R$.

2. *(Implementation to Protocol). The proof for the other direction is very similar.*

Going back to our example, we had the problem that under weak refinement, the *c?* call is reported as an error, as it is no longer accepted after *e?*. With strict-observational refinement, *e?* is no longer relevant for the refinement check between the student protocol and the participant behaviour. As a result, the only problem reported is the deadlock due to the second *c?* call shown in red in the participant behaviour of figure 5.12.

Thus as expected, strict-observational refinement does not imply weak refinement, as shown in the counter example in figure 5.12. However, although strict-observational refinement has been created with weak refinement as its basis, weak refinement also does not imply strict-observational refinement. This is due to the differences in handling of internal actions: While weak refinement allows suffixed internal actions, strict-observational does not. A counter example with a MIO B which is a weak, but not a strict-observational refinement of a MIO A is shown in figure 5.13.

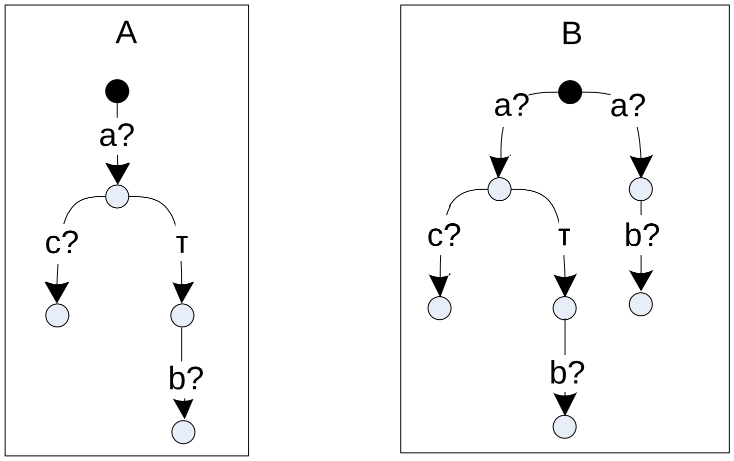


Figure 5.13: Weak vs. SO Refinement

The difference lies in the fact that weak refinement relates the state following the right-hand-side *a?* of automaton B with the state after `tau` in A. Strict-observational refinement, however, must choose the state following `a?` in A.

A remaining question is the relationship between strong refinement as defined in chapter 2, and strict-observational refinement. Here, it can be shown that strong implies strict-observational refinement.

Proposition 1 ***[Strong and Strict-Observational Refinement]*** *Let S and T be MIOs. If $S \leq_m T$, then also $S \leq_{so} T$.*

Proof 3 *of Prop. 1 (Strong Ref. implies Strict-Observational Ref.)*

Let $S \leq_m T$, we have to show that $S \leq_{so} T$. Because of $S \leq_m T$, we have a relation $R \subseteq states_S \times states_T$ as specified in $\leq_m$. We prove that R is also a relation for $S \leq_{so} T$.

First, $ext_T = ext_S$ as $S \leq_m T$. Thus, all actions not in ext_T are internal by definition (and furthermore, $int_T = int_S$). Now, R obviously includes $(start_S, start_T)$. We show (i) and (ii) from Definition 10 (Strict-Observational Modal Refinement) for all $(s,t) \in R$.

1. *(Protocol to Implementation). Assume $(s,t) \in R$ and*

   $$t \xrightarrow{a}_T^{\triangleleft(ext_T)} t'$$

   for some $a \in ext_T$. In more detail, this transition is written as

   $$t \xrightarrow{b_1}_T t_1 \dots t_{n-1} \xrightarrow{b_n}_T t_n \xrightarrow{a}_T t'$$

   with $b_1, \dots, b_n \in (int_T = int_S)$. For each of the transitions b_i, we can apply rule (1) from the definition of $\leq_m$, in each case moving one step in both S and T, finally reaching $(s_n, t_n) \in R$.

   From (s_n, t_n) we can again apply rule (1) with the external action a. We thus know that there exists s' with

   $$s_n \xrightarrow{a}_S s'$$

   and $(s', t') \in R$. Thus, we also have

   $$s \xrightarrow{a}_S^{\triangleleft(ext_T)} s'$$

   since we have only taken internal transitions not in ext_T followed by the final transition with $a \in ext_T$.

2. *(Implementation to Protocol). The proof for the other direction is very similar.*

5.2.2.2 Compatibility

Now we focus on our second aim which concerns compatibility checks. Consider figure 5.14, which is a replica of figure 5.12, however this time with complements of the protocols given before. The question is whether the protocols are compatible with their complements and whether the protocols are compatible with the participant behaviour.

Let us again check the example, this time focusing on the question of whether the required student protocol is compatible with the participant behaviour. We expect compatibility to hold; however, weak compatibility reports a similar violation as observed before during our discussion of refinement: After having taken the `e?` transition, which is external to the participant behaviour but not shared with the protocol, the *c!* transition of the protocol `P` can no longer be taken. This problem cannot be alleviated by hiding as hiding does not affect

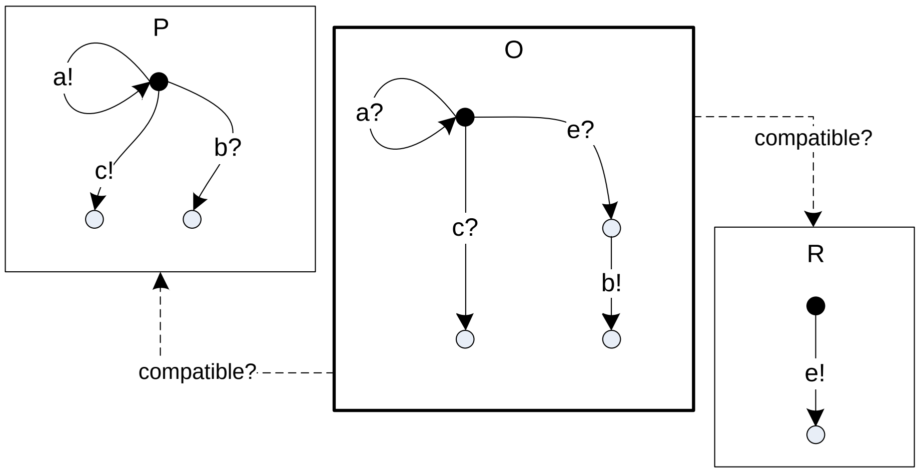


Figure 5.14: MIO Example for Strict-Observational Analysis (2)

weak modal compatibility in any relevant way. Instead, the idea is to reduce the set of state pairs considered during compatibility checking.

We want to consider both protocols to be compatible with the (new) service `O`, in each case only considering the protocol-observable actions as before. We define strict-observational I/O compatibility to reach this goal.

Definition 11 (Strict-Observational I/O Compatibility) *Let S and T be composable MIOs, and let L be the set $shared(S,T)$ of shared labels of S and T. S and T are called strict-observationally I/O compatible, written $S \sim_{so} T$, iff there exists a relation $R \subseteq states_S \times states_T$ containing $(start_S, start_T)$ such that for all $(s,t) \in R$,*

1. *for all $a \in (out_S \cap in_T)$, if $s \overset{a}{\dashrightarrow}{}_S^{\triangleleft(L)} s'$ then there exists $t' \in states_T$ such that $t \overset{a}{\longrightarrow}{}_T^{\triangleleft(L)} t'$ and $(s',t') \in R$,*
2. *for all $a \in (out_T \cap in_S)$, if $t \overset{a}{\dashrightarrow}{}_T^{\triangleleft(L)} t'$ then there exists $s' \in states_S$ such that $s \overset{a}{\longrightarrow}{}_S^{\triangleleft(L)} s'$ and $(s',t') \in R$.*

Considering again the example of compatibility between the participant behaviour and the student protocol, strict-observational I/O compatibility treats `e?` differently, as it is not defined in the student protocol — it is no longer relevant in its own right, but only as a prefix to `b!`. We therefore get a positive compatibility result between `O` and `P` as expected.

Lemma 3 *Strict-observational I/O compatibility $\sim_{so}$ is symmetric.*

As seen in the above example, strict-observational refinement does not imply weak refinement as expected (figure 5.14). The other direction, however, holds true; weak compatibility between two MIOs A and B implies strict-observational compatility.

Proposition 2 ***[Weak and Strict-Observational Compatibility]*** *Let S and T be composable MIOs. If $S \sim_{wc} T$, then also $S \sim_{so} T$.*

Proof 4 *of Prop. 2 (Weak Comp. implies Strict-Observational Comp.)*

Let $S \sim_{wc} T$, we have to show that $S \sim_{so} T$. Because of $S \sim_{wc} T$, we have a relation $R \subseteq states_S \times states_T$ as specified in $\sim_{wc}$. We prove that R is also a relation for $S \sim_{so} T$. First, R includes $(start_S, start_T)$. Second, we need to show (i) and (ii) from Definition 11 (Strict-Observational I/O Compatibility) for all $(s,t) \in R$; L being defined as $shared(S,T)$.

1. *(Protocol to Implementation). Assume $(s,t) \in R$ and*

$$s \overset{a}{\dashrightarrow}{}_S^{\triangleleft(L)} s'$$

for some $a \in (out_S \cap in_T)$. In more detail, this transition is written as

$$s \xrightarrow{b_1}{}_S s_1 \dots s_{n-1} \xrightarrow{b_n}{}_S s_n \xrightarrow{a}{}_S s'$$

with $b_1, \dots, b_n \in (act_S \cup act_T) \setminus shared(S,T)$. Since the transitions take place in S, we can reduce this to $int_S \cup ext_S \setminus shared(S,T)$. For each of the transitions b_i, we can apply rule (3) from the definition of $\sim_{wc}$, arriving at $(s_n, t) \in R$ (we are thus not moving in T).

For the final $a \in out_S \cap in_T$, we can employ rule (1) of $\sim_{wc}$ for $s_n \overset{a}{\dashrightarrow}_S s'$ and conclude that

$$t \xrightarrow{a}{}_T^{\triangleleft} t'$$

and $(s', t') \in R$. Obviously, we then also have

$$t \xrightarrow{a}{}_T^{\triangleleft(L)} t'$$

as the actions which can be bypassed in $\triangleleft(L)$ include int_S.

2. *(Implementation to Protocol). The proof for the other direction is very similar.*

5.2.2.3 Interface Theory

With strict-observational modal refinement and compatibility in place, we can now come back to our initial goal of defining a domain-specific interface theory targeted at checking protocol compliance in SOA systems.

In order to prove that MIOs together with strict-observational modal refinement and strict-observational I/O compatibility form an interface theory, we need an appropriate notion of composition for MIOs, called strict-observational composition and denoted by $\otimes_{so}$, which is adapted to our strict-observational view. Then we show that indeed $\mathcal{I}_{so} = (\mathcal{MIO}, \leq_{so}, \sim_{so}, \otimes_{so})$, where $\mathcal{MIO}$ is the domain of all modal I/O automata, satisfies preservation of compatibility and compositionality.

Definition 12 (Strict-Observational Composition) *Two composable modal input/output automata S_1 and S_2 can be strict-observationally composed to a MIO $S_1 \otimes_{so} S_2$ defined by*

- $states_{S_1 \otimes_{so} S_2} = states_{S_1} \times states_{S_2}$,
- $start_{S_1 \otimes_{so} S_2} = (start_{S_1}, start_{S_2})$,
- $in_{S_1 \otimes_{so} S_2} = (in_{S_1} \setminus out_{S_2}) \uplus (in_{S_2} \setminus out_{S_1})$,
- $out_{S_1 \otimes_{so} S_2} = (out_{S_1} \setminus in_{S_2}) \uplus (out_{S_2} \setminus in_{S_1})$,
- $int_{S_1 \otimes_{so} S_2} = \emptyset$.

The transition relations are given by

1. *for all* $a_i \in shared(S_1, S_2)$, $1 \leq i \leq n$, *if there exists* $c \in (ext_{S_1} \setminus shared(S_1, S_2))$ *and* $s_1 \overset{\lhd}{\xrightarrow{a_1}}_{S_1} \dots \overset{\lhd}{\xrightarrow{a_n}}_{S_1} s_1' \overset{\lhd}{\xrightarrow{c}}_{S_1} s_1''$, *and* $s_2 \overset{\lhd}{\xrightarrow{a_1}}_{S_2} \dots \overset{\lhd}{\xrightarrow{a_n}}_{S_2} s_2'$, *then* $(s_1, s_2) \xrightarrow{c}_{S_1 \otimes_{so} S_2} (s_1'', s_2')$ *(if $n = 0$ then $s_1 = s_1'$ and $s_2 = s_2'$),*
2. *for all* $a_i \in shared(S_1, S_2)$, $1 \leq i \leq n$, *if there exists* $c \in (ext_{S_2} \setminus shared(S_1, S_2))$ *and* $s_1 \overset{\lhd}{\xrightarrow{a_1}}_{S_1} \dots \overset{\lhd}{\xrightarrow{a_n}}_{S_1} s_1'$, *and* $s_2 \overset{\lhd}{\xrightarrow{a_1}}_{S_2} \dots \overset{\lhd}{\xrightarrow{a_n}}_{S_2} s_2' \overset{\lhd}{\xrightarrow{c}}_{S_2} s_2''$, *then* $(s_1, s_2) \xrightarrow{c}_{S_1 \otimes_{so} S_2} (s_1', s_2'')$ *(if $n = 0$ then $s_1 = s_1'$ and $s_2 = s_2'$),*
3. *and (iv), two similar rules for may-transitions ($\dashrightarrow$).*

The intuition of this composition operator is to capture non-shared external actions c, which may be prefixed by a number of shared actions $a_1 \dots a_n$ (or none at all): any paths only involving transitions with shared labels do not appear in the form of a series of synchronised actions in the composition; in fact, compositions $S_1 \otimes_{so} S_2$ do not contain any internal actions at all. In particular, any communication failures of two composed MIOs which are not compatible (i.e. a shared output is not received) do not emerge in the composed MIO.

Strict-observational composition satisfies associativity and commutativity which is a basic requirement for any reasonable composition operator (note that for commutativity to hold, we assume that two MIOs are considered equal if they have the same (internal and external) alphabet and there exists a bijection between the state spaces such that both may- and must-transition relations are preserved).

Lemma 4 *Strict-observational composition $\otimes_{so}$ is commutative and associative.*

Proof 5 *of Lem. 4 (Commutativity and Associativity of Composition) Commutativity is easy see as the definition of composition is symmetric. We focus here on the associativity of composition, i.e. to show that* $(S \otimes_{so} T) \otimes_{so} U = S \otimes_{so} (T \otimes_{so} U)$ *modulo state bijection. First, we need to show that that* S *is composable with* T *and* $S \otimes_{so} T$ *is composable with* U *if and only if* T *is composable with* U *and* S *is composable* $T \otimes_{so} U$*.*

Assume that S *is composable with* T *and* $S \otimes_{so} T$ *is composable with* U*. From the definition of composability, it follows that*

$$(in_S \cup int_S) \cap (in_T \cup int_T) = \emptyset \qquad (i)$$

$$(out_S \cup int_S) \cap (out_T \cup int_T) = \emptyset \qquad (ii)$$

$$[(in_S \setminus out_T) \cup (in_T \setminus out_S) \cup int_S \cup int_T \cup (out_S \cap in_T) \cup (out_T \cap in_S)] \\ \cap (in_U \cup int_U) = \emptyset \qquad (iii)$$

$$[(out_S \setminus in_T) \cup (out_T \setminus in_S) \cup int_S \cup int_T \cup (out_S \cap in_T) \cup (out_T \cap in_S)] \\ \cap (in_U \cup int_U) = \emptyset \qquad (iv)$$

By applying $(A \setminus B) \cup (A \cap B) = A$*, the equations* (iii) *and* (iv) *can be simplified.*

$$(in_S \cup in_T \cup int_S \cup int_T) \cap (in_U \cup int_U) = \emptyset \qquad (v)$$

$$(out_S \cup out_T \cup int_S \cup int_T) \cap (in_U \cup int_U) = \emptyset \qquad (vi)$$

Now, we have to show that T *is composable with* U*, i.e.*

$$(in_T \cup int_T) \cap (in_U \cup int_U) = \emptyset \qquad (vii)$$

$$(out_T \cup int_T) \cap (out_U \cup int_U) = \emptyset \qquad (viii)$$

Equation (vii) *follows directly from* (v)*, and* $(viii)$ *follows from* (vi)*. Next, we need to show that* S *is composable* $T \otimes_{so} U$*, i.e.*

$$(in_S \cup int_S) \cap (in_U \cup int_U \cup in_T \cup int_T) = \emptyset \qquad (ix)$$

$$(out_S \cup int_S) \cap (out_U \cup int_U \cup out_T \cup int_T) = \emptyset \qquad (x)$$

The left-hand side of (ix) *can be transformed as follows.*

$$\begin{aligned} & (in_S \cup int_S) \cap (in_U \cup int_U \cup in_T \cup int_T) \\ = \; & \underbrace{[(in_S \cup int_S) \cap (in_U \cup int_U)]}_{=\emptyset,\ \text{from } (i)} \cup \underbrace{[(in_S \cup int_S) \cap (in_T \cup int_T)]}_{=\emptyset,\ \text{from } (v)} \end{aligned}$$

Similarly, the left-hand side of (x) *can be transformed.*

$$\begin{aligned} & (out_S \cup int_S) \cap (out_U \cup int_U \cup out_T \cup int_T) \\ = \; & \underbrace{[(out_S \cup int_S) \cap (out_U \cup int_U)]}_{=\emptyset,\ \text{from } (ii)} \cup \underbrace{[(out_S \cup int_S) \cap (out_T \cup int_T)]}_{=\emptyset,\ \text{from } (vi)} \end{aligned}$$

Hence, S *is composable* $T \otimes_{so} U$*. The proof for the inverse direction can be shown in the same way.*

Now, as a small lemma, it is convenient to show that if S, T and U are composable, then $shared(S, T \otimes_{so} U) = shared(S,T) \uplus shared(S,U)$. By definition of shared and since S, T and U are composable, it holds that

$$shared(S, T \otimes_{so} U) = (in_S \cap out_{T \otimes_{so} U}) \uplus (in_{T \otimes_{so} U} \cap out_S)$$

This term can be further transformed as follows.

$$\begin{aligned} & (in_S \cap (out_T \setminus in_U \uplus out_U \setminus in_T) \uplus (out_S \cap (in_T \setminus out_U \uplus in_U \setminus out_T)) \\ = \; & (in_S \cap out_T \setminus in_U) \uplus (in_S \cap out_U \setminus in_T) \\ & \uplus (out_S \cap in_T \setminus out_U) \uplus (out_S \cap in_U \setminus out_T) \end{aligned}$$

This last term can be simplified again since S, T and U are composable and therefore $in_S \cap in_U = \emptyset$, $out_S \cap out_U = \emptyset$ $in_S \cap in_T = \emptyset$, and $out_S \cap out_T = \emptyset$:

$$(in_S \cap out_T) \uplus (in_S \cap out_U) \uplus (out_S \cap in_T) \uplus (out_S \cap in_U)$$

This is exactly the same as

$$shared(S,T) \uplus shared(S,U)$$

Finally, we show that

$$(s_S, (s_T, s_U)) \xrightarrow{c}_{S \otimes_{so} (T \otimes_{so} U)} (s'_S, (s'_T, s'_U))$$

if and only if

$$((s_S, s_T), s_U) \xrightarrow{c}_{S \otimes_{so} (T \otimes_{so} U)} ((s'_S, s'_T), s'_U)$$

Assume hence that

$$(s_S, (s_T, s_U)) \xrightarrow{c}_{S \otimes_{so} (T \otimes_{so} U)} (s'_S, (s'_T, s'_U))$$

The same must be shown for may also, but this can be shown just as below. We deal with each of the cases according to the definition of $\longrightarrow_{S \otimes_{so} (T \otimes_{so} U)}$.

(1) Let $c \in ext_S \setminus ext_{T \otimes_{so} U}$. By definition, there is a path $w_S \in (int_S \uplus shared(S,T) \uplus shared(S,U))^*$ in S to $\hat{s}_S$ with

$$\hat{s}_S \overset{\triangleleft}{\xrightarrow{c}}_S s'_S$$

Similarly, there is a path in $T \otimes_{so} U$, $w \subseteq w_S$, $w \in (shared(S,T) \uplus shared(S,U))^*$ from (s_T, s_U) to (s'_T, s'_U). Note that w does not contain any internal actions due to the definition of strict-observational composition. From the existence of the path w, it follows that there are paths $w_T \in (int_T \uplus shared(T,U) \uplus shared(T,S))^*$ from s_T to s'_T, and $w_U \in (int_U \uplus shared(U,T) \uplus shared(U,S))^*$ from s_U to s'_U. Both w_T and w_U correspond to w on shared actions. Now, considering (s_S, s_T) in $S \otimes_{so} T$, from the existence of w_S and w_T which correspond on shared actions, we can derive that there is a path from (s_S, s_T) to $(\hat{s}_S, s'_T)$, $w_{ST} \in (shared(T,U) \uplus shared(S,U))^*$. Furthermore, we know that $(\hat{s}_S, s'_T) \xrightarrow{c}_{S \otimes_{so} T} (s'_S, s'_T)$. As w_{ST} correspond to w_U on shared actions, we can derive that

$$((s_S, s_T), s_U) \xrightarrow{c}_{(S \otimes_{so} T) \otimes_{so} U} ((s'_S, s'_T), s'_U)$$

(2) Let $c \in ext_S \setminus ext_{T \otimes_{so} U}$. By definition, there is a path $w_S \in (int_S \uplus shared(S,T) \uplus shared(S,U))^*$ in S to s'_S. Similarly, there is a path in $T \otimes_{so} U$, $w \subseteq w_S$ $w \in (shared(S,T) \uplus shared(S,U))^*$ from (s_T, s_U) to $(\hat{s}_T, \hat{s}_U)$, and a transition

$$(\hat{s}_T, \hat{s}_U) \overset{\triangleleft}{\xrightarrow{c}}_{T \otimes_{so} U} (s'_T, s'_U)$$

Again, w does not contain any internal actions by definition of strict-observational composition. From the existence of the path w, it follows that there are paths $w_T \in (int_T \uplus shared(T,U) \uplus shared(T,S))^*$ from s_T to $\hat{s}_T$, and $w_U \in (int_U \uplus shared(U,T) \uplus shared(U,S))^*$ from s_U to $\hat{s}_U$. Again, both w_T and w_U correspond to w on shared actions. Now, considering (s_S, s_T) in $S \otimes_{so} T$, from the existence of w_S and w_T which correspond on shared actions, we can derive that there is a path from (s_S, s_T) to $(s'_S, \hat{s}_T)$, $w_{ST} \in (shared(T,U) \uplus shared(S,U))^*$. Again, it holds that w_{ST} corresponds to w_U on shared actions. Next, we need to consider the origin of $c \in ext_S \setminus ext_{T \otimes_{so} U}$.

(a) $c \in ext_T \setminus ext_U$. Here, we know that $\hat{s}_U = s'_U$, and by definition of strict observational composition also that $\hat{s}_T \overset{\triangleleft}{\xrightarrow{c}}_T s'_T$. Hence,

$$(s'_S, \hat{s}_T) \xrightarrow{c}_{S \otimes_{so} T} (s'_S, s'_T)$$

and therefore also $((s'_S, \hat{s}_T), \hat{s}_U) \xrightarrow{c}_{(S \otimes_{so} T) \otimes_{so} U} ((s'_S, s'_T), s'_U)$.

(b) $c \in ext_U \setminus ext_T$. Conversely, it holds that $\hat{s}_T = s'_T$, and $\hat{s}_U \overset{\triangleleft}{\xrightarrow{c}}_T s'_U$. From this, it follows directly that

$$((s'_S, \hat{s}_T), \hat{s}_U) \xrightarrow{c}_{(S \otimes_{so} T) \otimes_{so} U} ((s'_S, s'_T), s'_U)$$

Altogether, we have shown that

$$((s_S, s_T), s_U) \xrightarrow{c}_{(S \otimes_{so} T) \otimes_{so} U} ((s'_S, s'_T), s'_U)$$

The inverse direction can be proven in the same way, which concludes the proof for associativity of strict-observational composition.

First, we show that strict-observational compatibility is preserved under strict-observational modal refinement.

Theorem 1 [Preservation of Compatibility] *Let S, T, T' be MIOs, and let S, T and S, T' be composable. If $S \sim_{so} T$, $T' \leq_{so} T$ and $shared(S,T) = shared(S,T')$, then it follows that $S \sim_{so} T'$.*

Proof 6 *of Thm. 1 (Preservation of Compatibility)*

Assume $S \sim_{so} T$ and $T' \leq_{so} T$. We have to show that $S \sim_{so} T'$. From the first assumption it follows that there exists a strict-observational I/O compatibility relation R_C between S and T. From the second assumption it follows that *there exists a strict-observational modal refinement relation R_T for T' and T. We define a relation $R \subseteq states_S \times states'_T$ as follows:*

$$R = \{(s, t') \mid \exists\, t \in states_T.\ (t', t) \in R_T \wedge (s, t) \in R_C\}$$

We first show that $(start_S, start_{T'}) \in R$. As $S \sim_{so} T$, we know that $(start_S, start_T) \in R_C$. As $T' \leq_{so} T$, we know that $(start_{T'}, start_T) \in R_T$. It follows that $(start_S, start_{T'}) \in R$ from the definition of R.

Now, assume $(s, t') \in R$. We have to show (i) and (ii) from Definition 11 of strict-observational I/O compatibility. Let $L = shared(S, T) = shared(S, T')$.

1. *(S to T'). Assume*

$$s \overset{a!}{\dashrightarrow}{}_S^{\triangleleft(L)} \overline{s}$$

for some $a \in (out_S \cap in_{T'})$. Per definition of R, there exists a $t \in T$ with $(t', t) \in R_T$ and $(s, t) \in R_C$. Because $(s, t) \in R_C$ and $s \overset{a!}{\dashrightarrow}{}_S^{\triangleleft(L)} \overline{s}$ we know that

$$t \xrightarrow{a?}{}_T^{\triangleleft(L)} \overline{t} \text{ and } (\overline{s}, \overline{t}) \in R_C.$$

- *If also $t \xrightarrow{a?}{}_T^{\triangleleft} \overline{t}$ then because of $(t', t) \in R_T$ it follows $t' \xrightarrow{a?}{}_{T'}^{\triangleleft(ext_T)} \overline{t}'$ and $(\overline{t}', \overline{t}) \in R_T$. By Lemma 2 also $t' \xrightarrow{a?}{}_{T'}^{\triangleleft(L)} \overline{t}'$ since $L \subseteq ext_T$. Per definition of R we get $(\overline{s}, \overline{t}') \in R$.*
- *Otherwise there exists $b_i \in ext_T \setminus L$, $1 \leq i \leq n$, such that*

$$t \xrightarrow{b_1}{}_T^{\triangleleft} \dots \xrightarrow{b_n}{}_T^{\triangleleft} \hat{t} \xrightarrow{a?}{}_T^{\triangleleft} \overline{t}.$$

By a stepwise application of refinement we get

$$t' \xrightarrow{b_1}{}_{T'}^{\triangleleft(ext_T)} \dots \xrightarrow{b_n}{}_{T'}^{\triangleleft(ext_T)} \hat{t}' \xrightarrow{a?}{}_{T'}^{\triangleleft(ext_T)} \overline{t}'$$

such that $(\overline{t}', \overline{t}) \in R_T$. Similar to above (since $L \subseteq ext_T$), by Lemma 2 we get

$$t' \xrightarrow{b_1}{}_{T'}^{\triangleleft(L)} \dots \xrightarrow{b_n}{}_{T'}^{\triangleleft(L)} \hat{t}' \xrightarrow{a?}{}_{T'}^{\triangleleft(L)} \overline{t}'$$

and because $b_i \notin L$ it follows that $t' \xrightarrow{a?}{}_{T'}^{\triangleleft(L)} \overline{t}'$.

2. *(T' to S). Assume*

$$t' \overset{a!}{\dashrightarrow}{}_{T'}^{\triangleleft(L)} \overline{t}'$$

for some $a \in (out_{T'} \cap in_S)$. Per definition of R, there exists a $t \in T$ with $(t', t) \in R_T$ and $(s, t) \in R_C$.

- *If also $t' \overset{a!}{\dashrightarrow}{}_{T'}^{\triangleleft(ext_T)} \overline{t}'$ then we get $t \overset{a!}{\dashrightarrow}{}_T^{\triangleleft} \overline{t}$ and $(\overline{t}', \overline{t}) \in R_T$. By Lemma 2 also $t \overset{a!}{\dashrightarrow}{}_T^{\triangleleft(L)} \overline{t}$. By compatibility it follows $s \xrightarrow{a?}{}_S^{\triangleleft(L)} \overline{s}$, $(\overline{s}, \overline{t}) \in R_C$, and hence $(\overline{s}, \overline{t}') \in R$.*

- *Otherwise there exist $b_i \in ext_T \setminus L$, $1 \leq i \leq n$, such that*

$$t' \overset{b_1}{\dashrightarrow}{}_{T'}^{\triangleleft(ext_T)} \ldots \overset{b_n}{\dashrightarrow}{}_{T'}^{\triangleleft(ext_T)} \hat{t}' \overset{a!}{\dashrightarrow}{}_{T'}^{\triangleleft(ext_T)} \overline{t}'.$$

By a stepwise application of refinement we get

$$t \overset{b_1}{\dashrightarrow}{}_{T}^{\triangleleft} \ldots \overset{b_n}{\dashrightarrow}{}_{T}^{\triangleleft} \hat{t} \overset{a!}{\dashrightarrow}{}_{T}^{\triangleleft} \overline{t}$$

such that $(\overline{t}', \overline{t}) \in R_T$. By Lemma 2,

$$t \overset{b_1}{\dashrightarrow}{}_{T}^{\triangleleft(L)} \ldots \overset{b_n}{\dashrightarrow}{}_{T}^{\triangleleft(L)} \hat{t} \overset{a!}{\dashrightarrow}{}_{T}^{\triangleleft(L)} \overline{t}$$

and because of $b_i \notin L$, $t \overset{a!}{\dashrightarrow}{}_{T}^{\triangleleft(L)} \overline{t}$. By compatibility it follows $s \xrightarrow{a?}{}_{S}^{\triangleleft(L)} \overline{s}$, $(\overline{s}, \overline{t}) \in R_C$, and hence $(\overline{s}, \overline{t}') \in R$.

Compositionality is the prerequisite for independent implementability of services and their modular verification.

Theorem 2 [Compositionality] *Let S, T, T' be MIOs, and let S, T and S, T' be composable. If $T' \leq_{so} T$ and $shared(S, T) = shared(S, T')$, then $S \otimes_{so} T' \leq_{so} S \otimes_{so} T$.*

Proof 7 *of Thm. 2 (Compositionality of Refinement)*
Let S, T, T' be MIOs with $T' \leq_{so} T$. From $T' \leq_{so} T$ it follows that there exists a strict-observational refinement R_T for T' and T. We have to prove that $S \otimes_{so} T' \leq_{so} S \otimes_{so} T$ holds. Therefore, we define a relation $R \subseteq (states_S \times states_{T'}) \times (states_S \times states_T)$ by

$$R = \{((s, t'), (s, t)) \mid (t', t) \in R_T\}$$

which we show now is a strict-observational refinement relation for $S \otimes_{so} T'$ and $S \otimes_{so} T$.

We immediately know by definition that $((start_S, start_{T'}), (start_S, start_T)) \in R$ holds. We now take an arbitrary pair $((s, t'), (s, t)) \in R$, and show (i) from Definition 10 of strict-observational modal refinement; condition (ii) can be proved in an analogous way.

Let

$$(s, t) \xrightarrow{a}{}_{S \otimes_{so} T}^{\triangleleft} (\overline{s}, \overline{t})$$

for some $a \in ext_{S \otimes_{so} T}$ (we only consider external actions as per definition of strict-observational modal refinement; note also that $ext_{S \otimes_{so} T}$ does not contain shared labels). We have to show that there exists

$$(s, t') \xrightarrow{a}{}_{S \otimes_{so} T'}^{\triangleleft(ext_{S \otimes_{so} T})} (\overline{s}, \overline{t}')$$

such that

$$((\overline{s}, \overline{t}'), (\overline{s}, \overline{t})) \in R.$$

In $S \otimes_{so} T$, all occurring actions are those which were already external in either S or T, i.e. not shared between S and T. Thus, either

(1) $a \in ext_S, a \notin ext_T$, *or*

(2) $a \notin ext_S, a \in ext_T$.

We deal with each of these two cases.

(1) $a \in ext_S, a \notin ext_T$
Given is the following:

$$(s,t) \xrightarrow{a}{}^{\triangleleft}_{S \otimes_{so} T} (\bar{s},\bar{t})$$

Using the Definition 12 of $\otimes_{so}$*, part (i), we get*

$$s \xrightarrow{b_1}{}^{\triangleleft}_{S} \dots \xrightarrow{b_n}{}^{\triangleleft}_{S} \hat{s} \xrightarrow{a}{}^{\triangleleft}_{S} \bar{s} \quad \text{and} \quad t \xrightarrow{b_1}{}^{\triangleleft}_{T} \dots \xrightarrow{b_n}{}^{\triangleleft}_{T} \bar{t}$$

for some $n \geq 0$*; either* $n = 0$ *then* $s = \hat{s}$ *and* $t = \bar{t}$*, or* $n > 0$ *then there exist shared actions* $b_1, \dots, b_n \in shared(S,T)$*. As said before,* $a \in ext_S, a \notin ext_T$*. Now, by induction on the length* $n \geq 0$*, we show that from the latter and the assumption* $(t', t) \in R_T$*, we can get*

$$t' \xrightarrow{b_1}{}^{\triangleleft(ext_T)}_{T'} \dots \xrightarrow{b_n}{}^{\triangleleft(ext_T)}_{T'} \bar{t}' \text{ and } (\bar{t}, \bar{t}') \in R_T.$$

Base case $n = 0$*. Then* $t = \bar{t}$*,* $t' = \bar{t}'$ *and* $(\bar{t}', \bar{t}) = (t', t) \in R_T$ *by assumption. Induction step. Assume that we have for some* n

$$t \xrightarrow{b_1}{}^{\triangleleft}_{T} \dots \xrightarrow{b_n}{}^{\triangleleft}_{T} t_n \xrightarrow{b_{n+1}}{}^{\triangleleft}_{T} \bar{t} \text{ and } (t', t) \in R_T.$$

By applying our induction hypothesis, we get

$$t' \xrightarrow{b_1}{}^{\triangleleft(ext_T)}_{T'} \dots \xrightarrow{b_n}{}^{\triangleleft(ext_T)}_{T'} t'_n \text{ and } (t'_n, t_n) \in R_T.$$

From strict-observational modal refinement, we get that

$$t'_n \xrightarrow{b_{n+1}}{}^{\triangleleft(ext_T)}_{T'} t'_{n+1} \text{ and } (t'_{n+1}, t_{n+1}) \in R_T$$

which concludes the proof by induction. Thus, we have shown that

$$t' \xrightarrow{b_1}{}^{\triangleleft(ext_T)}_{T'} \dots \xrightarrow{b_n}{}^{\triangleleft(ext_T)}_{T'} \bar{t}' \text{ and } (\bar{t}, \bar{t}') \in R_T.$$

When transforming this trace to action-weak transitions w.r.t. $\triangleleft(ext_{T'})$*, every transition with label* b_i *is preceded by finitely many transitions labelled with actions in* $ext_{T'} \setminus ext_T$ *which by assumption (*$shared(S,T) = shared(S,T')$*) are not in* $shared(S,T')$*. Hence by Definition 12 of strict-observational composition, we can successively compose these tranformed transitions with the transitions*

$$s \xrightarrow{b_1}{}^{\triangleleft}_{S} \dots \xrightarrow{b_n}{}^{\triangleleft}_{S} \hat{s} \xrightarrow{a}{}^{\triangleleft}_{S} \bar{s}$$

yielding a transition with label a in $ext_{S\otimes_{so}T'}$ preceded by finitely many transitions labelled with actions in $ext_{T'} \setminus ext_T$. Thus, by restricting the relevant actions to $ext_{S\otimes_{so}T}$, we get

$$(s,t')\xrightarrow{a}{}^{\triangleleft(ext_{S\otimes_{so}T})}_{S\otimes_{so}T'}(\bar{s},\bar{t}').$$

As we also have $(\bar{t}',\bar{t}) \in R_T$, we get

$$((\bar{s},\bar{t}'),(\bar{s},\bar{t})) \in R.$$

(2) (case $a \notin ext_S, a \in ext_T$).

Again, assume

$$(s,t)\xrightarrow{a}{}^{\triangleleft}_{S\otimes_{so}T}(\bar{s},\bar{t})$$

Using the Definition 12 of $\otimes_{so}$, part (i), we get

$$s\xrightarrow{b_1}{}^{\triangleleft}_{S}\dots\xrightarrow{b_n}{}^{\triangleleft}_{S}\bar{s} \quad \text{and} \quad t\xrightarrow{b_1}{}^{\triangleleft}_{T}\dots\xrightarrow{b_n}{}^{\triangleleft}_{T}\hat{t}\xrightarrow{a}{}^{\triangleleft}_{T}\bar{t}$$

for some $n \geq 0$; either $n = 0$ then $s = \hat{s}$ and $t = \bar{t}$, or $n > 0$ then there exist shared actions $b_1, \dots, b_n \in shared(S,T)$. One can show, similar to the previous case, that it holds (by induction) that there exist

$$t'\xrightarrow{b_1}{}^{\triangleleft(ext_T)}_{T'}\dots\xrightarrow{b_n}{}^{\triangleleft(ext_T)}_{T'}\hat{t}'\xrightarrow{a}{}^{\triangleleft(ext_T)}_{T'}\bar{t}' \text{ and } (\bar{t},\bar{t}') \in R_T.$$

Again, by the same argument as above, we get

$$(s,t')\xrightarrow{a}{}^{\triangleleft(ext_{S\otimes_{so}T})}_{S\otimes_{so}T'}(\bar{s},\bar{t}').$$

As we also have $(\bar{t}',\bar{t}) \in R_T$, we get

$$((\bar{s},\bar{t}'),(\bar{s},\bar{t})) \in R.$$

The second part (ii) ($\leq_{so}$, Implementation to Protocol) can be proven in a similar way as above, just with may-transitions instead of must-transitions.

Independent implementability is a direct consequence of preservation of compatibility under refinement and compositionality of refinement. Even more than in traditional software architectures, SOA systems benefit from this property due to the inherent distribution of services in the service-based application landscape.

Corollary 1 [Independent Implementability] *Let S, T, T' be MIOs, and let S, T and S', T' be composable. If $S \sim_{so} T$, $T' \leq_{so} T$, $S' \leq_{so} S$ and $shared(S,T) = shared(S',T')$, both $S' \sim_{so} T'$ and $S' \otimes_{so} T' \leq_{so} S \otimes_{so} T$ follow.*

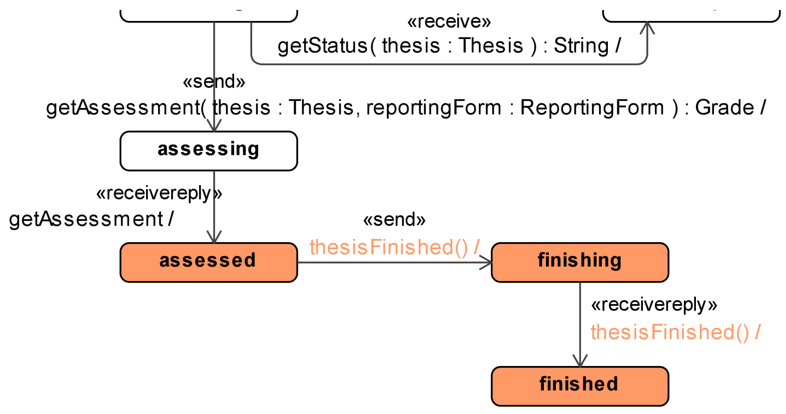


Figure 5.15: Incorrect Excerpt from Case Study (2)

Proof 8 *of Corollary 1 (Independent Implementability) First, it can be easily shown that*

$$shared(S, T) = shared(S, T') = shared(S', T').$$

By Thm. 1, it follows $S \sim_{so} T'$, *and by Lem. 3,* $T' \sim_{so} S$. *Again by Thm. 1 and Lem. 3, it follows that* $S' \sim_{so} T'$. *Second, we show* $S' \otimes_{so} T' \leq_{so} S \otimes_{so} T$. *We apply Thm. 2 to get* $S \otimes_{so} T' \leq_{so} S \otimes_{so} T$. *Moreover, by Thm. 2, it follows* $T' \otimes_{so} S' \leq_{so} T' \otimes_{so} S$. *Since* $\otimes_{so}$ *is commutative (Lem. 4) and* $\leq_{so}$ *transitive (Lem. 1), it holds that* $S' \otimes_{so} T' \leq_{so} S \otimes_{so} T$.

With preservation of compatibility under refinement and compositionality (which together imply independent implementability), we have arrived at our goal of defining an interface theory for strict-observational analysis of MIOs.

Corollary 2 [Strict-Observational Interface Theory] $\mathcal{I}_{so} = (\mathcal{MIO}, \leq_{so}, \sim_{so}, \otimes_{so})$ *is an interface theory.*

5.2.2.4 Example

Let us consider again our (adapted) case study in figure 5.11 on page 163. Running strict-observational refinement on this example will show *no errors*: The race condition is ignored, and as there is no additional deadlock, the analysis completes sucessfully.

For an actual error, consider a revised `tutor` protocol as shown in figure 5.15 (note that this is not a required protocol of the `ThesisManagement` participant, but the opposite version of the `tutor` service). The new version of the protocol has an added exchange between `tutor` and `ThesisManagement`, in which the tutor is informed about and has to acknowledge the official end of the thesis.

Checking the participant behaviour against this new `tutor` protocol with *weak compatibility* (as usual, with hiding) yields the same problem as with *weak* *refinement* as expected and discussed in the last section. The higher-level problem of the complete `thesisFinished` call being missing from the participant behaviour is not reported, as it comes after the race condition.

Checking the participant behaviour with *strict-observational compatibility*, on the other hand, yields the desired result: The `thesisFinished` exchange is flagged as not being possible in the participant behaviour after the `getAssessment` call.

5.2.3 Conclusion

In this section, we have discussed exploiting the modal input/output representation of UML4SOA via the Service Meta-Model (SMM) for formal analysis. In particular, we have discussed two of many possible options for analysing MIOs: Firstly, we have discussed using the existing notions of *weak* modal refinement and compatibility, which report all kinds of deadlocks including those created by race conditions. Secondly, we have introduced *strict-observational* refinement and compatibility, a new domain-specific interface theory which is explicitly targeted at verifying protocol-based compliance of participant behaviour and protocols, i.e. detecting deadlocks which are not based on race conditions.

We believe that with these two verification options, UML4SOA modellers gain the ability to verify their models early in the development process. Both options are tool-supported as we shall discuss in the next section, and can be used prior to generating code as discussed in chapter 6.

5.3 Tool Support

This chapter has presented formal support for UML4SOA: Firstly, we have discussed a semantics for UML4SOA models — both participant behaviour and protocols — given in denotational style with the semantic domain of modal I/O transition systems. Secondly, we have discussed analysis of the modelled SOA system on the MIO level, and hinted at the relevance the results of this analysis have on the UML level.

In this section, we introduce and discuss tool support for formal analysis of UML4SOA models. Enabling such analysis requires three steps:

- First, we translate UML4SOA models into modal I/O transition systems. This has been discussed in section 5.1.

- Such a transformation enables analysis to be performed on the MIO level, for example using the interface theory described in section 5.2.

- Results of the analysis can be readily shown on the MIO level; however, to allow the use of the formal analysis by developers not knowledgeable or interested in these details, a better way of visualising them is a back-annotation on the UML4SOA level. This requires a third step.

We have implemented these three steps in two tools. The first tool, called the *Mio Workbench*, is a general-purpose editor and verification tool for modal I/O automata. It implements various interface theories and thus notions of refinement and compatibility, among them the strict-observational interface theory introduced in section 5.2. This tool takes care of the second step in the above process; it is described first as the other steps use some of the core technology introduced in the workbench. The workbench is discussed in section 5.3.1.

The second tool, called *UtbM* (UML4SOA to-and-back-from MIOs) covers transforming UML4SOA models to MIOs as discussed in section 5.1. Furthermore, it is able to annotate the analysis results from the Mio Workbench back to UML4SOA models (*back-annotation*). The tool integrates with both a UML modelling tool (MagicDraw) and the Mio Workbench to fully automate this process. UtbM is discussed in section 5.3.2.

5.3.1 The Mio Workbench

The background chapter as well as section 5.2 have introduced modal I/O automata as well as interface theories with different notions of refinement and compatibility. While the definitions given there as mathematical formula are concisely formulated and unambiguous, manually checking an automata for a certain property is tedious and error-prone. This holds especially true if, as in our case, the automata are automatically generated from UML models. We have therefore implemented a mechanical checker able to analyse MIOs based on different interface theories. The resulting software system, called the *Mio Workbench*, is an Eclipse-based verification tool and editor for modal I/O automata.

In itself, the Mio Workbench is independent of UML4SOA, as it can be used on arbitrary modal I/O automata and with arbitrary notions of refinement and compatibility.

5.3.1.1 Features

The most direct and intuitive way to work with MIOs is using a graphical editing facility based on a graph of nodes (states) and edges (transitions) as well as accompanying labels. The first feature provided by the workbench is thus an editor:

- *Graphical Editor*, allowing creation of new or changing existing MIOs.

The implementation of the different notions of refinement and compatibility are the next features of the Mio Workbench:

- *Refinement Verification.* These include strong, may-weak, weak, and strict-observational modal refinement.
- *Compatibility Verification.* We support the notions of strong (with and without "helpful" environment, cf. [dAH05]), weak, and strict-observational modal compatibility.

Furthermore, the Mio Workbench supports actual composition of composable MIOs:

- *Composition Operations* on MIOs (standard and strict-observational).

The output of a composition operation is either the composed MIO or a list of problematic actions which caused the composition to fail.

Considering refinement and compatibility verification, we can get two very important, but very different results. First, if refinement or compatibility is possible, we get refinement relation(s) and matching states, respectively. However — and this is even more important — if the verification fails, we get the error states and the error transitions in the two automata, i.e. the exact position(s) which led to the erroneous outcome.

During out work, we have noticed that having a visual feedback when testing different notions of refinement and compatibility is of great help to understanding the subtleties involved in the definitions. Therefore, the workbench also includes:

- *Refinement relation and state match view.* If a refinement or compatibility verification was successful, the workbench graphically displays the relation or the matching states side-by-side between the two input MIOs.

- *Problem view including error states and unmatched actions.* If a refinement or compatibility verification was not successful, the workbench graphically displays, side-by-side, the path which led to an erroneous state, and the transition possible in one automaton, but not in the other.

5.3.1.2 Architecture

On the technical side, the Mio Workbench is based on the Eclipse platform. We use an Eclipse Modeling Framework (EMF)-based meta-model for MIOs, which enables persistence and simple access to concrete automata. The workbench integrates into Eclipse by adding MIO-specific file handling and the new MIO editor as well as the verification view. The Mio Workbench is extensible with regard to new notions of refinement, compatibility, and composition, by means of standard Eclipse extension points. The architecture of the Mio Workbench is shown in figure 5.16.

The figure shows the Mio Workbench component with its three main subcomponents — the MIO meta-model, which uses the Eclipse EMF meta-model, and the editor and operation view UI elements, which contribute to the Eclipse workbench. Finally, the Mio Workbench contributes the MIO file handling to the Eclipse resource environment.

The underlying framework for all data operations in the Mio Workbench is the EMF-based meta-model of modal I/O automata shown in figure 5.17. A `ModalIOAutomaton` (bottom) includes a set of states, may- and must transitions as well as input, output, and internal actions. The start `State` is denoted with a special association. A transition is linked to two states (source and target) via

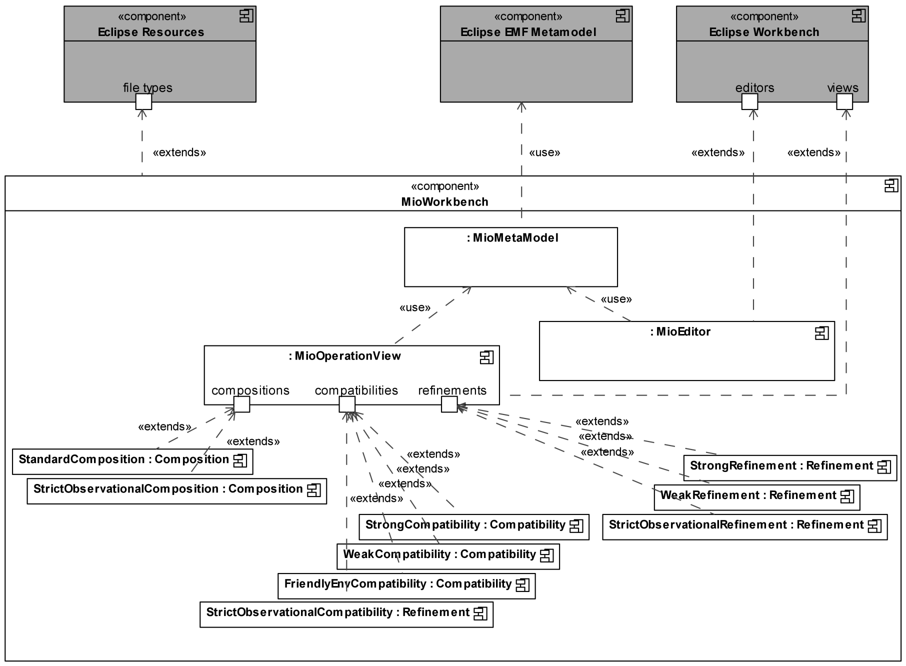

«component»
Eclipse Resources
file types
«component»
Eclipse EMF Metamodel
«component»
Eclipse Workbench
editors
views
«extends»
«use»
«extends»
«extends»
«component»
MioWorkbench
: MioMetaModel
«use»
«use»
: MioEditor
: MioOperationView
compositions
compatibilities
refinements
«extends»
«extends»
StandardComposition : Composition
StrictObservationalComposition : Composition
StrongCompatibility : Compatibility
WeakCompatibility : Compatibility
FriendlyEnvCompatibility : Compatibility
StrictObservationalCompatibility : Refinement
StrongRefinement : Refinement
WeakRefinement : Refinement
StrictObservationalRefinement : Refinement

Figure 5.16: Mio Workbench: Architecture

the abstract `Transition` superclass of may- and must transitions. Likewise, it is linked to an action represented with the abstract `Action` superclass of input, output, and internal actions.

Within the Mio Workbench components, the operation view and the editor both use the Mio Model. The most interesting component is, of course, the operation view, which allows executing operations such as compatibility checks in the workbench. Various refinements, compatibility notions and compositions are provided per default in the Mio Workbench (all shown in the figure). As the Mio Workbench has an open architecture, additional notions of refinement, compatibility and composition can be plugged in without changing the code.

5.3.1.3 User Interface

Figure 5.18 shows the MIO editor inside the Eclipse workbench with the (incorrect) eUniversity participant behaviour from the last section. On the left-hand side, the project explorer shows MIOs stored on the file system as .mio files; on the right-hand side, the editor for one of these MIOs is displayed. A MIO is displayed in the classical way by using nodes as states and edges as transitions. Each transition has a type (must or may), which is indicated by a square or diamond, respectively. Furthermore, each transition also stands for an internal, input, or output action. An input action is coloured green and is suffixed with

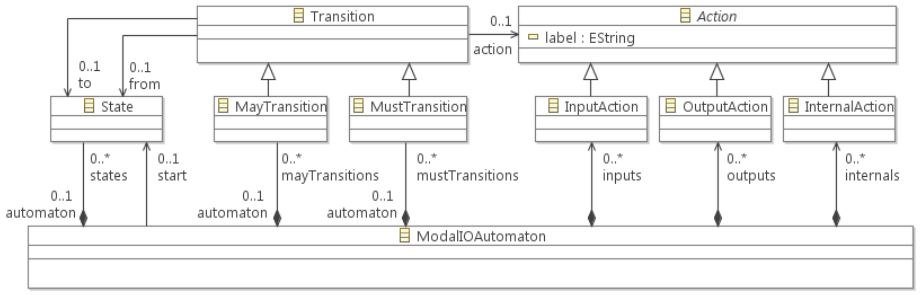
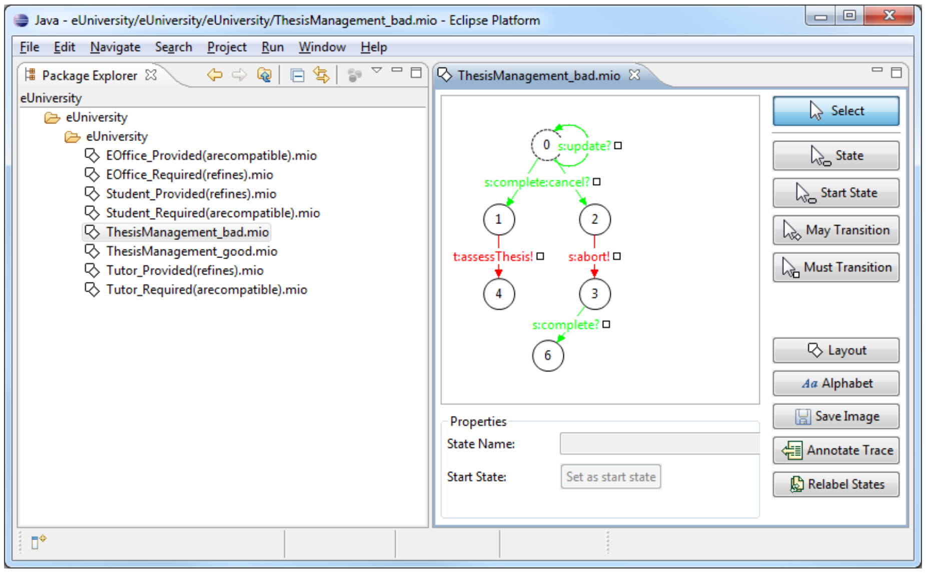


Figure 5.17: MIO Meta-Model

a question mark (*?*). An output action is coloured red and is suffixed with an exclamation mark (*!*). Finally, an internal action is gray and does not have a suffix. The MIO editor offers all the usual operations such as adding new nodes, moving them around, changing labels, types, and re-layouting.


Java - eUniversity/eUniversity/eUniversity/ThesisManagement_bad.mio - Eclipse Platform

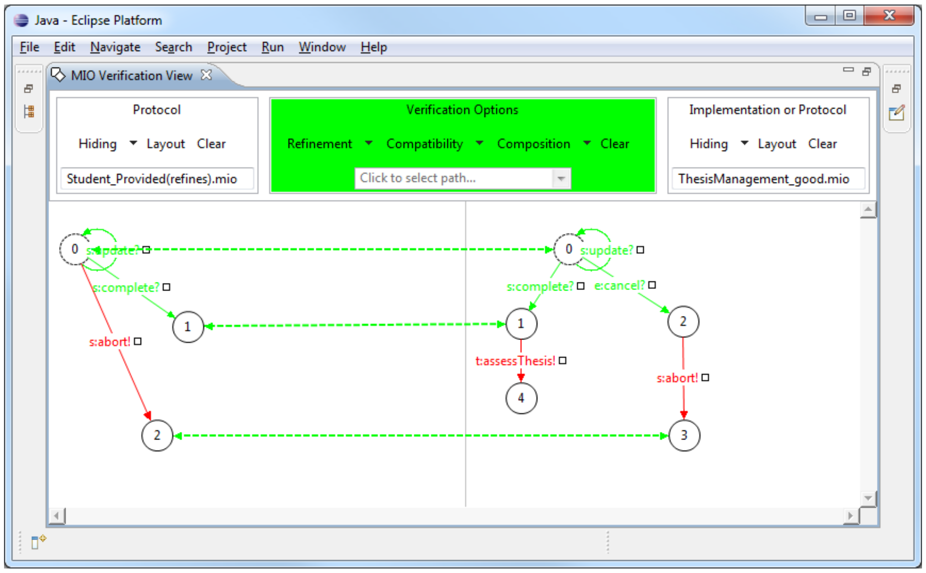
eUniversity
eUniversity
eUniversity
EOffice_Provided(arecompatible).mio
EOffice_Required(refines).mio
Student_Provided(refines).mio
Student_Required(arecompatible).mio
ThesisManagement_bad.mio
ThesisManagement_good.mio
Tutor_Provided(refines).mio
Tutor_Required(arecompatible).mio

Properties
State Name:
Start State: Set as start state

Figure 5.18: Mio Workbench: Editor

The verification view of the Mio Workbench is the central access point to the verification functionality. It features a side-by-side view of two modal I/O automata, which can then be analysed for refinement or compatibility, or composed.

Figure 5.19 shows an example of a successful refinement check: The (correct) eUniversity participant behaviour from the example in the last section is shown to be a refinement of the student protocol. The green background in the top-

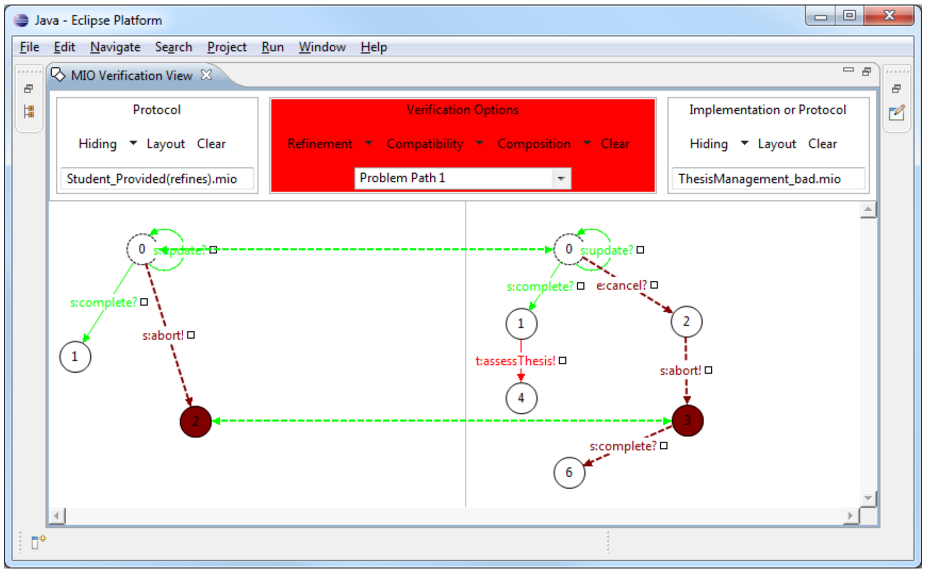


Student_Provided(refines).mio

Click to select path...

ThesisManagement_good.mio

Figure 5.19: Mio Workbench: Refinement View



Student_Provided(refines).mio

Problem Path 1

ThesisManagement_bad.mio

Figure 5.20: Mio Workbench: Refinement Problem View

centre panel indicates a successful check; in the side-by-side view below, green dashed arrows indicate which states are in relation.

As noted above, the most interesting results are negative cases, i.e. if a refinement does not exist or compatibility does not hold. In this case, the MIO Workbench displays the possible error paths, each indicating a state pair in violation and the corresponding erroneous action.

Figure 5.20 shows the visualisation of a strict-observational refinement again, but this time with the incorrect participant behaviour from the previous section. This participant behaviour is not a refinement of the student protocol, which is indicated by the red top-centre panel and the fact that two states are marked red. These states form the state pair in which one action is possible in one automaton (*s:complete?* on the right) but not in the other.

The Mio Workbench contains additional helpful features such as automatically laying out MIOs, adjusting an alphabet of a MIO by hiding non-shared labels for a compatibility or refinement check, and more.

5.3.1.4 Summary

In this section, we have presented a verification tool and graphical editor for modal I/O automata called the Mio Workbench, which implements various refinement and compatibility notions based on MIOs. We believe that tool support is of great help for discussing modal I/O automata and may serve in research, teaching, and as a prototype for industrial applications. The Mio Workbench is open source and can be freely downloaded from `http://www.miowb.net`.

5.3.2 UtbM: From UML4SOA to MIOs and Back

Section 5.1 has presented an abstract view on the transformation algorithm we use for defining modal I/O transition systems as the formal semantics of UML4SOA participant behaviour and protocols. In this section, we discuss tool support — the UtbM transformer tool — for generating MIOs from directly from the UML4SOA syntax, and for back-annotating results of analyses in the Mio Workbench to UML. In principle, the UML2MIO part of UtbM is a model transformation tool: It works on UML4SOA models, transforming them to MIO models. Both input and output models are based on the Eclipse Modelling Framework (EMF). The transformation itself follows the algorithms discussed in section 5.1. UtbM is implemented as a hybrid MagicDraw and Eclipse plug-in in Java.

5.3.2.1 Features

The UtbM tool has two objectives, and therefore, two features:

- UML to MIO. UtbM allows conversion of (graphical) UML models modelled in a standard UML modelling tool into MIOs.

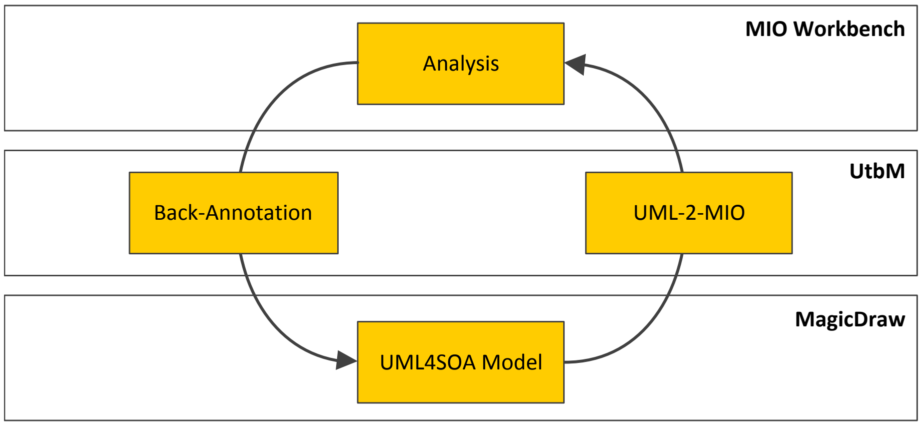


Figure 5.21: The Workflow of Using UtbM

- MIO to UML. The reverse step is not a transformation but a back-annotation, which allows annotating the result of an analysis operation in the Mio Workbench into the graphical representation of the UML model.

The process of using these features in combination with a modelling tool and the Mio Workbench is shown in figure 5.21.

For the purpose of directly working with graphical UML models, UtbM integrates into the UML modelling tool MagicDraw, which is used as a front-end to the entire process of converting to MIOs, analysing, and re-annotating the result in UML. Furthermore, the first feature is also available in Eclipse, allowing developers to directly work with both UtbM and the Mio Workbench.

5.3.2.2 Architecture

The architecture of the UtbM tool is shown in figure 5.22. The figure shows the UtbM system at the bottom, including three sub-systems called UserInterface, UML2MIO, and Results2UML.

UtbM has two user interface components. The first integrates into the MagicDraw platform, enabling developers to transform to MIOs and get their back-annotation on the graphical models in UML. The second integrates into Eclipse, allowing developers to work on the actual source XMI and MIO files with more fine-grained control over the analysis in the Mio Workbench.

The UML2MIO transformer is based on the Eclipse UML meta-model as indicated above, plus the MIO meta-model defined as part of the Mio Workbench.

Finally, the Results2UML back-annotation is based on both the results of the MIO Workbench and on the MagicDraw platform to enable drawing results in the original UML diagrams.

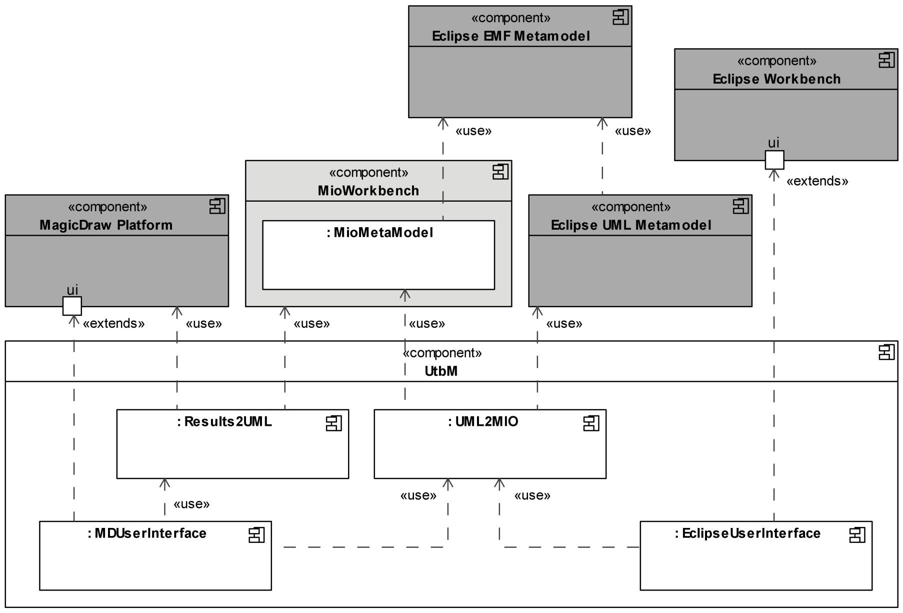


Figure 5.22: UtbM: Architecture

5.3.2.3 User Interface

The goals of the two user interface components in UtbM is rather different: While the integration into MagicDraw aims at ease of use for the developers, the integration into Eclipse takes a hands-on approach to the full analysis details.

Integration into MagicDraw is shown in figure 5.23. Within the graphical view, a developer selects a protocol and a participant behaviour or two protocols and selects a check method in the (new) UML4SOA menu. The result is shown in the same diagrams by means of colour: As in the Mio Workbench, red signals an error; green signals success.

Integration into Eclipse is shown in figure 5.24. The transformation process is started by right-clicking an UML2-EMF-XMI file and selecting the appropriate transformation entry. A dialog is displayed with activities and PrSMs to choose from; the result is shown directly in the editor of the Mio Workbench.

5.3.2.4 Summary

In this section, we have introduced UtbM, a model transformation and back-annotation tool which bridges the gap between UML models designed with the help of the UML4SOA profile and their semantic domain of modal I/O automata. The first part of the tool, UML2MIOs, implements the semantics described in detail in section 5.1 and can be invoked from both MagicDraw and within Eclipse. The second part, Results2UML, enables users to view the

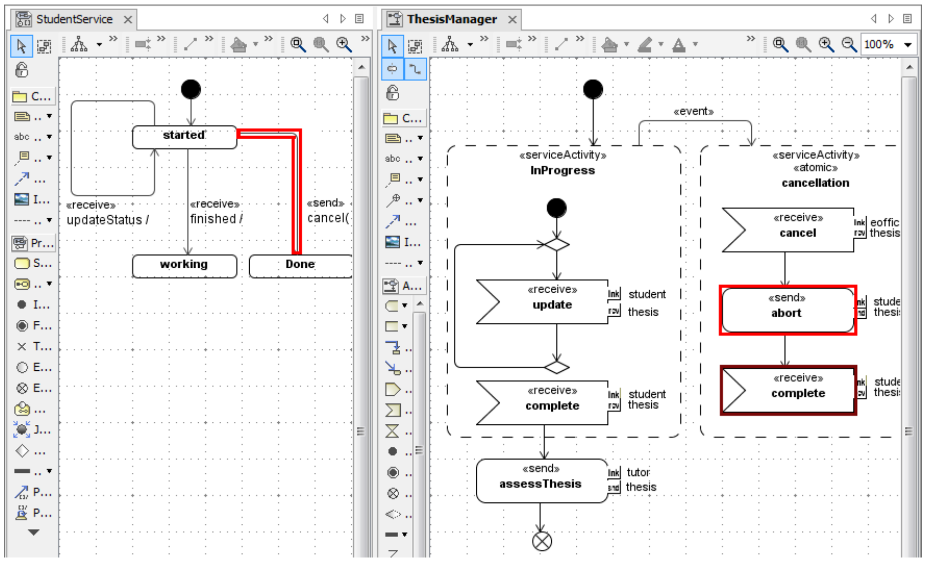
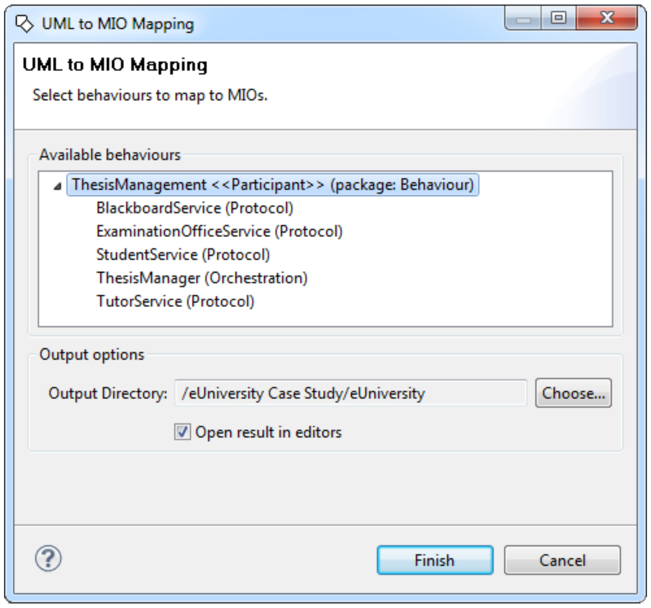


Figure 5.23: UtbM: MagicDraw Integration


UML to MIO Mapping

UML to MIO Mapping

Select behaviours to map to MIOs.

Available behaviours

- ThesisManagement <<Participant>> (package: Behaviour)
  - BlackboardService (Protocol)
  - ExaminationOfficeService (Protocol)
  - StudentService (Protocol)
  - ThesisManager (Orchestration)
  - TutorService (Protocol)

Output options

Output Directory: /eUniversity Case Study/eUniversity

☑ Open result in editors

Figure 5.24: UtbM: Eclipse Integration

analysis results of the Mio Workbench directly on the graphical models within MagicDraw.

5.4 Related Work

The analysis approach discussed in this chapter revolves around the idea of *protocol analysis*, i.e. determining whether a certain behavioural specification matches its specified protocols. A new interface theory has been presented which contributes a high-level view on this problem. Related work for this theory is discussed in section 5.4.1.

Furthermore, this chapter has introduced the Mio Workbench, a verification tool built on the domain of modal I/O automata and interface theories. See section 5.4.2 for related work for the workbench.

Finally, there are other forms of analysis available for UML4SOA which we shall discuss in section 5.4.3.

5.4.1 Protocol-Based Analysis

This section is concerned with related work for our definition of an interface theory for protocol-based analysis of modal input/output automata. The general study of interface theories was started by de Alfaro and Henzinger in [dAH01b, DHJP08]. Their well-known interface theory called interface automata essentially builds on the formalism of input/output automata [LT87, LT89, GL00] accompanied with notions of refinement (defined by alternating simulation) and compatibility [dAH01a]. This theory has recently been generalised to an interface theory based on modal input/output transition systems [LNW07a, LNW07b] which uses modal automata [LT88a, HL89] for modelling interface behaviour.

However, less attention has been paid to refinement between interfaces with different alphabets. Recently, the approach proposed in [RBB$^+$09a, RBB$^+$09b] deals with alphabet extension by adding self-loops for the new actions to all states of the automaton. It is easy to see (same argument as given in the previous section) that this solution is not adequate to handle our situation.

In [FBU09] Fischbein et al. propose branching alphabet refinement of modal transition systems to cope with the problem of unintuitive implementations allowed by weak modal refinement. However, their refinement is classic in the sense that it considers single transitions in the preconditions, which is too strict for our application area.

Action refinement [GR01] is a flexible notion of refinement which is based on refining actions when changing the abstraction level: An action of an abstract specification can be decomposed into a sequence of low-level actions specifying the system in more detail. We differ from this notion as all of our actions reside on the same abstraction level, yet we only consider some of them depending on the current viewpoint.

There is also an extensive body of knowledge on analysis of Web Service orchestrations based on industry standards like BPEL; [tBBG07] provides a decent overview. However, to the best of our knowledge, no approach so far has considered early application-level verification as a precursor to existing approaches.

Instead, the focus lies on analysis of specific aspects of service orchestrations. Both [LMSW06] and [Mar05] analyse BPEL compositions through transformations to petri-nets. Their composition analysis assumes a friendly environment in the sense of [dAH01a], but is not geared towards application-level analysis of service orchestrations.

Fu et al. [FBS04] present a translation of BPEL processes to Promela, the input language of the SPIN model checker [Hol03]. However, due to the interaction semantics of the translation, application-level verification is not feasible.

Two further approaches that are closer to ours use calculi to specify the underlying labelled transition systems. [BZ07] explicitly focuses on strong notions of compliance and compatibility; for calculi-based model-checking approaches like [FGL$^+$08], the same reasoning as for [FBS04] applies: application-level verification is prohibited by the composition semantics of the language.

5.4.2 Tooling

The Mio Workbench is a verification tool for modal input/output automata; however, with the semantics presented for UML4SOA we can also characterise it as a formal analysis tool for SOAs.

Regarding the first view, the Mio Workbench differs from existing tools by explicitly focusing on both modality and input/output aspects of MIOs. In fact, we believe that it is the first verification workbench with this functionality. Related work in this section include the Modal Transition System Analyser (MTSA) [DFCU08, DFFU07] which uses modal transition systems; TICC [AdAdS$^+$06] (for input/output transition systems), and finally Tempo [LMS08], which is based on timed input/output automata.

Compared to SOA verification tools such as WS-Engineer [FUMK07], the Mio Workbench is further removed from the input language and thus requires a dedicated back-annotation tool (such as UtbM).

Furthermore, the Mio Workbench is more experimental in nature: Based on input given directly as MIOs, it allows different (pluggable) interface theories to be applied to investigate both the model and the theories themselves.

5.4.3 Other Analysis Methods

During the development of UML4SOA, two approaches to the formal verification of service models have been particularly inspiring to the author. The first addresses quantitative analysis of service models; the second deals with qualitative analysis for interacting service processes. We shall provide pointers to these two approaches in the following.

First, an important aspect of systems based on service-oriented architectures are scalability issues, as the systems are inherently distributed and in many cases offer an interface to customers via the Internet. Qualitative analysis of service models can provide insights into the performance of SOA systems. The PEPA stochastic process calculus [Hil96] provides different means of performing such analysis. A mapping from UML4SOA to PEPA is available in executable form. PEPA allows both discrete- and continuous-state interpretations, which enables the goal of scalable analysis of scalable systems [CGT09b].

A discussion of using PEPA on UML4SOA models may be found in [TG10], which introduces an end-to-end example of a quantitative analysis using PEPA on one of the case studies of the Sensoria project.

Second, the previous sections have provided an analysis approach for protocol verification of service behaviour. A different approach is provided by WS-Engineer [FUMK03, FUMK06], a verification tools which directly works with the *behaviours* of different services implemented in BPEL. Though not directly usable on UML4SOA models, the model transformers which will be introduced in chapter 6 and integrated using the tooling platform discussed in chapter 8 allow the exploitation of WS-Engineer for UML4SOA.

WS-Engineer provides several options for checking behavioural service specifications. In the context of UML4SOA service specifications, interaction checking of collaborating BPEL orchestrations is of particular interest, as these orchestrations can be generated from UML4SOA models. Another option in WS-Engineer is performing design analysis of BPEL processes against message sequence charts. In this scenario, model checking is used to verify that a BPEL process provides the necessary activities to meet the MSC specification.

More information on how WS-Engineer, with the starting point of UML4-SOA, can be used for model verification is provided in [FM08].

5.5 Summary

In this chapter, we have introduced and discussed formal support for UML4SOA with applications and tool support.

We have begun with a formal semantics for UML4SOA via the Service Meta-Model (SMM) by defining the algorithm used for transforming between SMM models and Modal I/O Automata (MIOs) both for participant behaviours and protocols. The denotational function has been described for each of the relevant elements of the SMM, given by the textual notation defined in chapter 4.

This denotational definition of the mapping between SMM models and modal I/O automata rigorously defines the *meaning* of participant behaviours and protocols and serves as the semantics of the SMM and the UML4SOA profile (chapter 3).

Having a representation of UML4SOA models as MIOs opens up a number of verification and checking options. In this thesis, we have focused on using *interface theories* to check refinement of an (implementation) MIO to a (protocol) MIO, and the compatibility of two (protocol) MIOs. Besides the option of using standard interface theories such as *strong* and *weak*, we have contributed our own option, the *strict-observational* interface theory, which is suitable for detecting deadlocks which are not race conditions. This interface theory can be used very early in the modelling phase to find and correct errors in the communication between services.

Furthermore, we have discussed tool support for the formal semantics of UML4SOA as well as verification on the MIO level. First, we have introduced the Mio Workbench, a formal verification tool built on the domain of modal I/O automata and supporting a set interface theories for verification of refinement and compatibility.

Second, we have discussed the UtbM tool, which implements the formal semantics given for UML4SOA, thus enabling automatic transformation between UML4SOA models and MIOs. The tool also supports *back-annotation*, i.e. showing the results of MIO-based analyses on a UML level.

Finally, we have discussed related work to the strict-observational interface theory as well as tool support in section 5.4.

Chapter 6

Transformations and Code

A key requirement of model-driven development (MDD) [Sel03] approaches is the ability to generate executable code from the model of a system. This can be achieved by using model transformations [MCG05] and model-to-code emitters to produce code in executable target languages. This chapter introduces model transformations with tool support for SOA systems based on UML4SOA models.

To illustrate the conversion of UML4SOA models to executable software artefacts, we provide two examples of model transformations and code generation with different targets. The first target is the Web Service (WS) standards family [WCL$^+$05] (which includes the Business Process Execution Language (BPEL) [OAS07] and surrounding artefacts such as WSDL [CCMW01] descriptions and XML Schema [FW04] types); the second target is the standard object-oriented programming language Java [GJSB05].

The basic ideas and intuition behind model transformation and code generation based on UML4SOA is discussed in section 6.1. As will be seen in this section, the Service Meta-Model (SMM) already introduced in chapter 4 is used as the basis for these transformations: We first create SMM instances out of UML4SOA models, and then move on to code.

The initial step of parsing UML4SOA models into instances of the SMM is discussed in section 6.2. The next two sections each discuss a transformation from the SMM to a model of an executable language: Transformation from the SMM to the Web Services family is discussed in section 6.3; SMM to Java is discussed in section 6.4.

We discuss tool support in section 6.5, followed by related work in section 6.6. Finally, a conclusion is drawn in section 6.7.

Published results: Results presented in this chapter are based on publications [KMH$^+$07], [MSK08b], [MSK08a], [FGK$^+$10a], and [GGK$^+$10]. Furthermore, the code generators are a result developed as an answer to one of the main Sensoria research objectives (model transformations) and has been reported in several technical reports, brochures, and presented at fairs.

6.1 Transforming Service Models

In this chapter, we discuss the transformation of UML4SOA-based models of SOA software with regard to two different targets: The first target is the Web Services family of standards (and in particular, the BPEL language); the second is the well-known Java programming language. Web Services and Java have been chosen as they are representations of two separate programming paradigms: Service- and object-orientation. The corresponding transformations and results are thus different in both structure and behaviour, and provide interesting insights into the mapping from UML4SOA to executable code.

The Business Process Execution Language (BPEL), introduced in 2003, is a language which has already been designed with services in mind, and can be regarded as the current industry standard for writing orchestrations based on *Web Services* technology. BPEL is platform-dependent in the sense of a close integration with the Web Service technology stack; furthermore, BPEL uses an XML-based syntax which is hard to read and write by hand, requiring even native BPEL editors to transform their own (mostly graphical) models of BPEL to code.

Java, on the other hand, has been introduced in 1995 and is one of the most successful object-oriented programming languages, widely used in both academia and industry to develop a wide range of software systems from small embedded programs to enterprise-level SOA solutions. Java does not include service concepts as first-level citizens. Any attempt to capture the service-based ideas of UML4SOA thus requires a careful mapping of SOA concepts to Java code, using as much as possible of the underlying Java infrastructure but introducing an additional (SOA) layer where appropriate.

As already introduced in chapter 2, *model transformations* based on meta-models of the source and target language are a core enabling technology in model-driven development approaches. In this chapter, we introduce model transformations between several different meta-models: Firstly, the UML meta-model with extensions from SoaML and UML4SOA; secondly, the SMM meta-model, and thirdly, the various meta-models of the Web Service family languages and again a single meta-model for Java.

6.1.1 From UML4SOA to Code

For developers, code generation in MDD4SOA should start with UML, SoaML and UML4SOA and directly lead to executable code. As pointed out in chapter 4, however, SoaML and UML4SOA are in fact the concrete syntax of the Service Meta-Model (SMM), which defines the structure of the SOA system and includes a rigorous semantics for the behavioural specifications.

Our model transformations to executable code should thus not directly start at the UML model, but should instead be based on the Service Meta-Model. This necessitates an initial parsing step from UML, SoaML, and UML4SOA to the SMM. As both ends of this parsing process are in fact meta-models, we have implemented this process already as a model transformation.

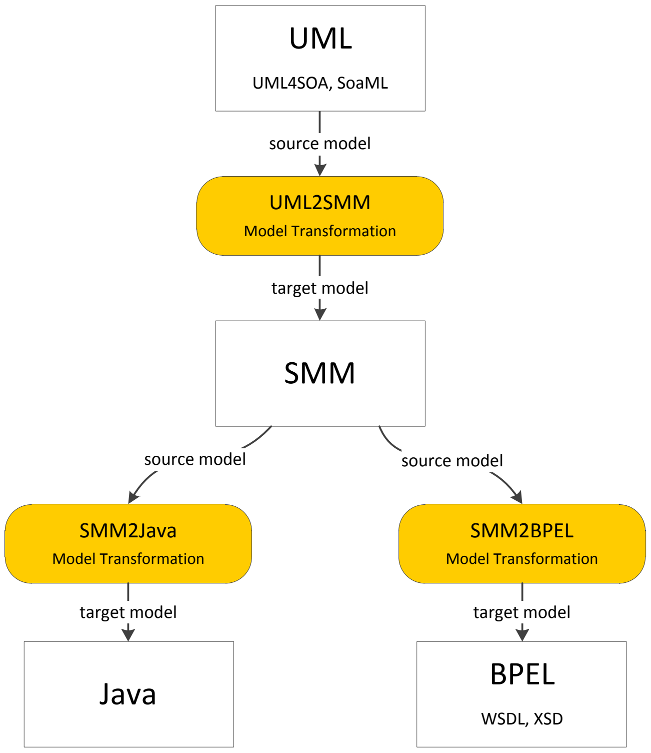


Figure 6.1: Two-Step Transformation Using the SMM

From the SMM, we can move on to our original targets, which is again implemented by model transformations: Meta-models exist for both the Web Services family standards and Java; our own meta-model for the SMM has already been defined in chapter 4. After each transformation, the generated instance of the target model can be serialised down to actual code, which can be readily executed.

Thus, the transformation of UML4SOA to code consists of two steps. As a first step, the UML model (including SoaML and UML4SOA extensions) is transformed (*parsed*) into an instance of the SMM; the second step consists of individual transformations from the SMM to actual code in a concrete target platform. The complete approach is shown in figure 6.1.

6.1.2 Key Aspects

Each of the three transformations shown in the figure must consider all the aspects present in a SOA system which have been introduced in chapter 4: Static aspects, behaviour, and data handling.

- *Static Aspects.* The root elements of SMM models are SOA participants.

These include provided and required services as the provided and required functionality of the actual behaviour of a participant. These services, in turn, are typed with service interfaces specifying the operations and parameters for invoking service functionality. Parameters and return types may be typed with primitive types or more complex message types which must then be considered as well (including their possible relationships).

- *Behaviour.* Participants may have one or more behaviours attached. The transformations need to consider the individual aspects of these behaviours: First, this includes structural concepts such as decisions, loops, parallel behaviour, exception handling, and the SOA-specific concepts of event and compensation handling. Secondly, we need to handle communication actions (for example, for sending and receiving calls); finally, other primitive actions like compensation calls or throwing exceptions need to be considered.
- *Data Handling.* Finally, a key aspect of executable services is data. The Service Meta-Model includes data manipulation elements which allow declaring variables, assignments, and primitive operations; all of these elements are well-typed. Depending on the source and target of the transformation, these elements need to be converted appropriately.

In the following, we first present the UML2SMM transformation, followed by the two transformations to executable target languages (SMM2WS, SMM2Java).

6.2 UML to SMM

The UML to SMM transformation handles the task of converting a complete UML/SoaML/UML4SOA model to an instance of the SMM, which includes the static, behavioural, and data aspects of the model discussed in section 6.1. The following sections contain a description of the individual transformations applied to the source elements of the UML model. The aim is to provide a conceptual overview of the transformation; technical details present in the full implementation are left out. For the complete executable transformation, see section 8.3.

The UML meta-model consists of 264 classifiers with a total of 618 structural features. For the UML to SMM transformation, however, not all of these elements are relevant; the transformation can be restricted to meta-classes and structural features allowed by UML4SOA/Strict and their attached stereotypes defined by SoaML and UML4SOA.

6.2.1 Parsing UML

Intuitively, creating an SMM instance out of SoaML and UML4SOA models is straightforward; however, some difficulties arise in practise which must be dealt with in the UML2SMM transformation.

- *Profile Element Recognition.* A first problem to be dealt with is the non-explicit representation of profile concepts in the UML meta-model. As discussed in chapter 3, UML profiles are not a first-level extension mechanism. As a consequence, the source model — UML — does not contain meta-classes from SoaML and UML4SOA, but instead tags existing meta-classes with stereotypes. Writing a transformation based on such a model thus requires matching stereotypes and classes, which is a tedious middle step required of all transformations based on UML profiles.
- *Workflow interpretation.* Another problem is the structural mismatch between UML activities and the SMM. Although UML4SOA/Strict already requires proper nesting of UML constructs such as decision and merge nodes, an activity model still (and purposefully) has a workflow intuition: A decision node, for example, can both be a branch start or a loop end depending on the context, while branches and loops are quite different concepts and are represented as such in the SMM and many standard programming languages. Furthermore, even though well-nesting is a requirement for UML4SOA diagrams, premature path termination by means of flow or activity final nodes and exceptions is commonly modelled without an explicit closing node; such situations need to be properly recognised when parsing the input model.
- *Parsing Data Statements.* Finally, UML4SOA introduces a data manipulation language with its own set of parsing rules and semantics. Being based on SoaML message types — which are UML classes — a transformation needs to ensure type correctness, and create a proper representation of the language constructs before converting the statements to the respective target model. This task thus requires the implementation of a parser for the data handling semantics.

In the following, we discuss the implementation of the UML2SMM transformation which addresses these concerns.

6.2.2 Transforming the Static Part

The roots of a SoaML/UML4SOA model of a SOA system are participants, i.e. UML classes stereotyped with ≪*Participant*≫. UML4SOA/Strict requires that the ports of the UML class are stereotyped with either ≪*Service*≫ or ≪*Request*≫; furthermore, each port type must be a class stereotyped with ≪*ServiceInterface*≫ which contains the operations provided or required at the given port.

An interface type may contain two types of operations: implemented and used ones. Declared operations of the UML ≪*ServiceInterface*≫ — inherited or declared directly on the type — are attached as implemented operations. Used operations are treated in a slightly different way: In UML, a class may declare its use of another class or interface, but not (without specifying behaviour) of the actual operations used. The UML2SMM transformation is therefore also restricted to usage relationships and adds all operations of used interfaces or types as used operations.

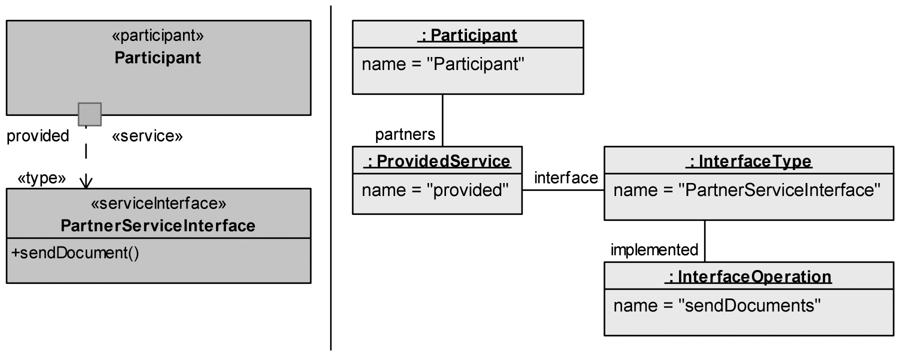


Figure 6.2: UML2SMM: Participants and Interfaces

An example for this first transformation step is shown in figure 6.2. Note that as the SMM has no graphical representation, the right-hand side is shown as an UML instance diagram of the SMM elements and their relationships.

Transformation 1 (Participants and Interfaces) *Each UML class stereotyped with ≪Participant≫ is transformed to an SMM `Participant` with the same name. The ports associated with the class are transformed to either SMM `ProvidedService` or `RequiredService` and are attached to the SMM `Participant`. For each port type, an instance of an SMM `InterfaceType` is created and set as the type of the service. As indicated above, operations implemented or used by the port type are both converted to `ServiceOperation` and attached using either the `used` or `provided` association.*

Parameters of interfaces are the first of many elements which require data types — either primitive or defined by the developer as a ≪*MessageType*≫. In UML4SOA/Strict, five primitive types may be used (`Integer`, `Double`, `Boolean`, `String`, and `Date`), which match the SMM versions; if one of these types is encountered in the input model, the corresponding SMM type is instantiated. Note that `Double` and `Date` are not native to the UML; however, many UML modelling tools include these types as extensions to the UML and thus can be readily used by developers.

Developers may also define their own types with associations and attributes. These are converted to SMM `MessageType` instances along with corresponding `MessageProperty` instances. Finally, exception types to be used in throw statements and exception handlers are converted to `ExceptionType` instances.

Types are encountered at several places in the input model — in parameters of service interfaces, properties of message types, and later on in variables, throw statements, and exception handlers. For each of these places, the following transformation step may be used. An example for this step is shown in figure 6.3

for two message types with an association between them and a reference to a primitive type.

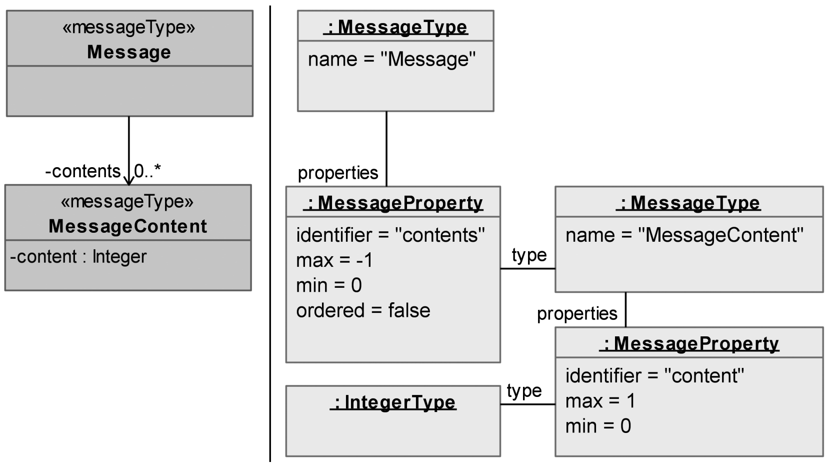


Figure 6.3: UML2SMM: Data Types

Transformation 2 (Data Types) *Each referenced primitive, message, or exception type in the UML source model is transformed to an instance of SMM `PrimitiveType`, `MessageType`, or `ExceptionType`. All outgoing associations of a message type are converted to `MessageProperty` instances with correct settings for `min`, `max`, and `ordered`. For the target of the association, the same rules apply.*

Declared or used operations may contain `in` and `return` parameters. Each parameter must be converted, along with its type, to instances of SMM `InterfaceParameter`. The next transformation step deals with such parameters. An example using an operation and a message type from the previous steps is given in figure 6.4.

Transformation 3 (Interface Parameters) *Each `in` parameter of an operation declared as part of a `MessageType` is converted to an SMM `InputParameter`; `min`, `max`, and `ordered` properties are set as appropriate. The `InputParameter` is added as a parameter of the operation. For `return` parameters, an SMM `OutputParameter` is created which is treated likewise.*

It is worth mentioning that the UML2SMM transformer creates a flat type structure, i.e. without packages and inheritance. However, any inheritance present in the input model is resolved; i.e., inherited associations and attributes are added to each type during conversion. This is mainly done for simplicity reasons and as a `MessageType`, being designed for network communication, normally does not rely on an intricate inheritance design. Furthermore, note that

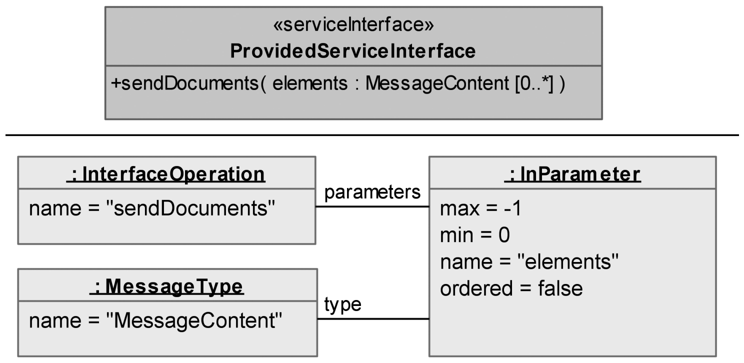


Figure 6.4: UML2SMM: Parameters of Service Operations

types are added on demand: Both primitive types of the UML and message types of the input model are only added to the SMM if they are actually used within the UML input model.

6.2.3 Handling UML4SOA Activities

A participant may own several activities, each realising a different implementation. In UML4SOA/Strict, these activities must be stereotyped with ≪*ServiceActivity*≫. The UML2SMM transformation converts each of these top-level activities to an SMM `ServiceActivity` which is attached to the participant using the `behaviours` association; afterwards, the (possibly nested) children of each UML service activity are transformed to the appropriate SMM constructs.

Before being able to create the appropriate SMM composite elements for an UML4SOA service activity, the inner *structure* of the activity must be recognised. As noted in the first section, an UML activity — even with the restrictions applied due to UML4SOA/Strict — has a workflow structure in which decision and merge nodes are used to create loops and decisions, the latter of which may or may not have a dedicated end merge node, as usage of raise and termination action is allowed, too. It is therefore not clear without further inspection which SMM composite elements to transform to.

To address this problem, the UML2SMM transformer uses a divide-and-conquer approach for transforming service activities. The first step follows the control flows of the activity, identifying groups of actions; the second uses this information to actually create SMM counterparts for the identified groups and actions. The two steps are detailed in the next two sections.

Transformation 4 (Divide-And-Conquer) *A group of UML elements is divided into partitions. Each partition is then conquered with an appropriate transformations step for the type of the partition. Note that the conquering part may recursively invoke step 4 for children of conquered partitions.*

The transformation starts by creating partitions for the root service activities of the participant and transforming each of them with transformation step 5. In turn, this step invokes the divide-and-conquer transformation step 4 for the children of the service activity.

6.2.3.1 Partitioning an Activity

The division step in the transformation from UML to the SMM requires a *partitioning* mechanism: The input elements are divided into individual *partitions* which correspond to either control constructs (activities, loops, branches, parallel behaviour) or individual actions. Partitioning is performed one layer after another: First, the root activity is partitioned; afterwards, the children of each recognised partition are partitioned, etc. This top-down process is repeated until only simple elements remain. There are five partition types, which can be characterised as follows:

- *Service Activity Partition.* A service activity partition is created if a nested service activity is encountered. Both the children of the partition as well as handlers attached to the activity are seen as part of the partition, as they need to be attached to the converted activity as well.

- *Loop Partition.* Loops in UML4SOA/Strict must have a dedicated start merge and final decision node with a link back from the final to the first node. If such a pattern is identified in the input activity, a loop partition is created and the inner elements of the loop are added to this partition.

- *Branch Partition.* A branch partition represents a decision in the input model, which is characterised by a decision start node with several outgoing guarded paths. Note that a final merge node might not exist, for example if one path ends with a termination or raises an exception. If such a pattern is identified in the input model, a branch partition is created. A branch imposes a certain semantics on its children: Only one of the *paths* which each represent a possible execution sequence is executed.

- *Parallel Partition.* For parallel behaviour, UML4SOA uses the standard fork and join nodes of UML; if these are encountered, a parallel partition is created. As in a branch, the children of a parallel partition are added to *paths.*

- *Single Action Partition.* Identifying and partitioning the input UML model sooner or later leads to the individual actions of UML4SOA, like send, receive, or data. For each of these actions, a new partition is created which marks the end of the nested structure.

An example of how the partitioning proceeds is shown in figure 6.5. The figure shows the same UML4SOA service activity three times. In the first example on the left, the overall process has been divided into three partitions. The upper and lower partition each consist of only one element and are thus *Single Action*

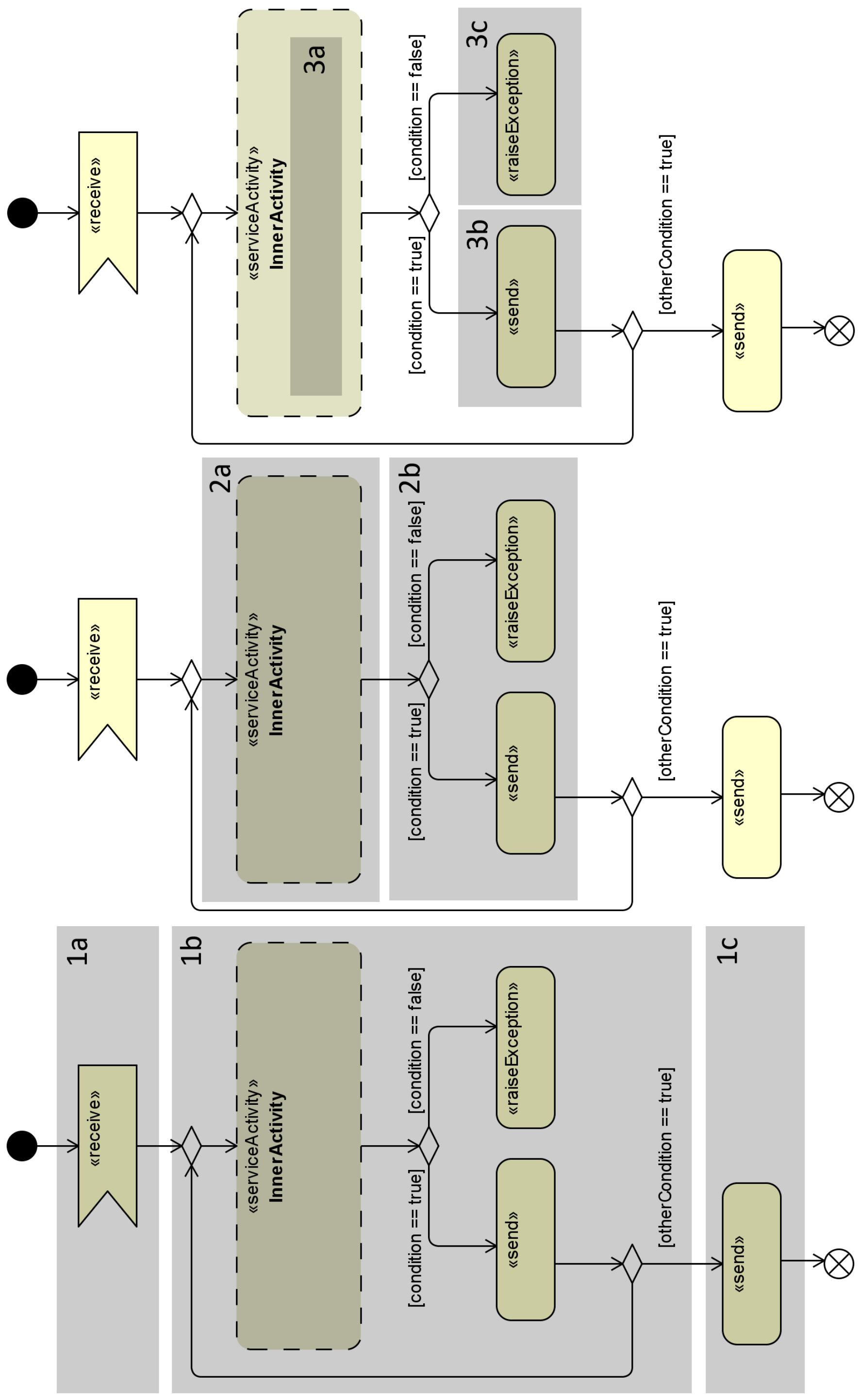


Figure 6.5: UML2SMM: Partitioning a Service Activity

Partitions (1a and 1c). In the middle, a loop has been found and thus, a *Loop Partition* (1b) has been created.

The loop partition is now further partitioned as shown in the process in the centre. In the upper part, a service activity is found and thus a *Service Activity Partition* (2a) is created. In the lower part, a decision has been identified which leads to a *Branch Partition* (2b).

In the last step, both the children of the *Service Activity Partition* and the *Branch Partition* are partitioned further (shown in the process to the right). The children of the service activity are omitted here (3a). In the lower part, the children of the branch are divided into two *Single Action Partitions* (3b and 3c) in separate paths of the branch.

Partitioning UML4SOA structures does not actively transform elements to the SMM, but is rather a preliminary phase to the actual conversion discussed in the next section. Nevertheless, it is interesting to have a look at the algorithm which is shown in listing 6.1. Its input is a set of UML activity nodes — this includes control nodes, data nodes, and executable nodes — and the control flows between them. One of the activity nodes is designated as the start node; in the example given above, this is the initial node. In the beginning of the behavioural transformation, the input elements used are the children of the root service activity; as the transformation progresses, children of inner service activities, loops, branches, and parallel blocks will be handled.

Starting from the initial node, the algorithm moves through the control flows between the given nodes. Depending on the type of node it encounters, different partitions are created. For merge, decision, and fork nodes, the algorithm performs a *look ahead* to identify the end of the partition. Everything in-between the current node and the identified end node of the partition is then added to the partition and the algorithm proceeds after the end node. Note that although the intuition of finding a corresponding end node of a partition is clear, the resulting algorithm is non-trivial; however, as it mainly deals with technical details it is not shown here.

Once the division algorithm has identified the (flat) list of partitions in the current set of activity nodes, this list is given to the conquering part of the transformation, which iterates through the partitions, creating SMM elements as appropriate or again turning to the division algorithm for a further division of the children of a partition. This is discussed in the next section.

6.2.3.2 Conquering Partitions

Having identified a list of partitions for the current level in the UML4SOA service activity, these partitions can be transformed to the appropriate SMM composite elements.

In the following, the partition types and their transformation to the SMM are discussed.

Listing 6.1: UML2SMM: Dividing Elements

```
Definition
    UML2SMM partitioning algorithm
    input: a set of UML activity nodes, their control flows,
            and a designated start node
    output: a list of partitions and their children

Algorithm
    create partition list
    current element := designated start node
    while current element is not null do
        if current is structured activity node then              ▷ service activity
            create service activity partition
                with children of service activity
            add handlers to partition
            add partition to partition list
            current element := next after current
        else if current is merge node then                                  ▷ loop
            look ahead to find corresponding decision node
            create loop partition with all elements until end node
            add partition to list
            current element := first after the decision node
        else if current is decision node then                             ▷ branch
            look ahead to find corresponding end (merge node, or last element)
            create branch partition
            for each outgoing link of decision node do
                create path with elements until end node
                add path to branch partition
            end for
            add branch to list
            current element := first after the end of branch
        else if current is fork node then                               ▷ parallel
            look ahead to find corresponding join node
            create parallel partition with these elements
            for each outgoing link of fork node do
                create path with elements until end node
                add path to parallel partition
            end for
            add parallel to partition list
            current element := first after join node
        else if current is service action then                            ▷ single
            create single action partition with this element
            add partition to partition list
            current element := next after current
        end if
    end while
    return partition list
```

Service Activity Partition

The first partition encountered in any transformation is the service activity partition, as each behaviour is in fact a (UML4SOA) service activity itself. A service activity is a grouping concept — it contains nested children with their own sequence of control flows. Furthermore, a service activity may have associations to handlers, which are service activities too and need to be added to the SMM. The following transformation step deals with service activities.

Transformation 5 (Service Activity Partition) *For each service activity partition, an SMM `ServiceActivity` is created and added to the `children` of the enclosing one (if one exists). For the children of the UML4SOA service activity, divide-and-conquer (step 4) is performed.*

Furthermore, all handlers of the UML service activity are transformed to instances of SMM `EventHandler`, `CompensationHandler`, or `ExceptionHandler` depending on the type. Each handler is added to the `handlers` association of the newly created SMM `ServiceActivity`. For the children of each handler, divide-and-conquer is performed. In the special case of an `ExceptionHandler`, the exception type of the caught exception is resolved to the corresponding SMM `ExceptionType` and added as the `exceptionType` of the handler.

Last, for each interrupting receive in the UML service activity (i.e., receives with an outgoing interrupting control flow), an SMM `Receive` (step 9) is created and added to the `interruptingReceives` association.

An example for a nested UML4SOA service activity and its corresponding SMM model is shown in figure 6.6. The service activity contains one event- and one compensation handler as well as one interrupting receive. Children of the activities are left out.

Branches, Loops, and Parallel Partitions

Besides service activity partitions, the other three main grouping concepts of UML4SOA are branches, loops, and parallel partitions. All three are similar in that they subject their children to a certain execution semantics: In a branch, only a certain set of children are executed; in a loop, the children are executed more than once; and in a parallel partition, certain sets of children are executed in parallel. In the SMM, these partitions are modelled as subclasses of `PathBasedPartition`; the sets of children are modelled by the `Path` class.

Transformation 6 (Branch) *For each branch partition, an SMM `Decision` instance is created. The children of the branch have already been separated into paths by the divide algorithm; for each path, an SMM `Path` instance is created and added to the `children` of the `Decision` instance. An UML4SOA decision has associated guards with each path; these guards are transformed to instances of `RightHandSideExpression` (discussed in the next section) which are added as the `enterCondition` of the corresponding `Path`. Finally, divide-and-conquer is performed for the children of each path.*

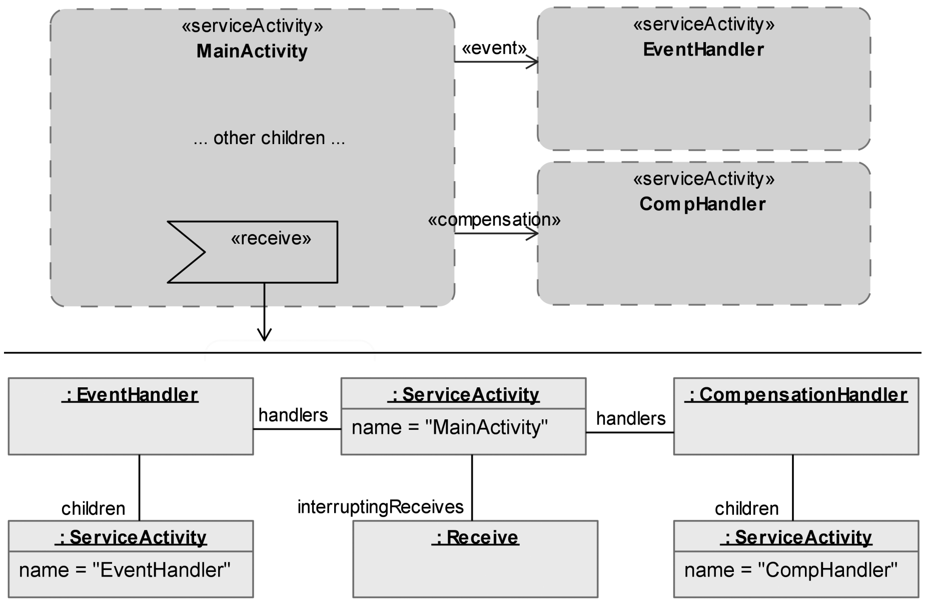


Figure 6.6: UML2SMM: Service Activities and Handlers

Transformation 7 (Loop) *For each loop partition, an SMM `Loop` instance is created. A loop instance only has one path which is transformed to an SMM `Path` and added to the `children` of the `Loop` instance. Furthermore, an UML4-SOA/Strict loop has a condition for exiting the loop; the condition is transformed to an instance of `RightHandSideExpression` (discussed in the next section) and added as the `leaveCondition` of the `Loop`. Finally, divide-and-conquer is performed for the children of the loop.*

Transformation 8 (Parallel) *For each parallel partition, an SMM `Parallel` instance is created. The children of the parallel partition have already been separated into paths by the divide algorithm; for each path, an SMM `Path` instance is created and added to the `children` of the `Parallel` instance. Finally, divide-and-conquer is performed for the children of each path.*

An example for a branch partition is given in figure 6.7. The example shows how one `Decision` is created for the branch partition in UML4SOA; each path is converted to a `Path` with (in this case) one child `Send`. Furthermore, a `RightHandSideExpression` with the condition is attached as the `enterCondition` of each path; for more details on expressions, see the next section.

As the conversion of loop and parallel is similar, examples are omitted.

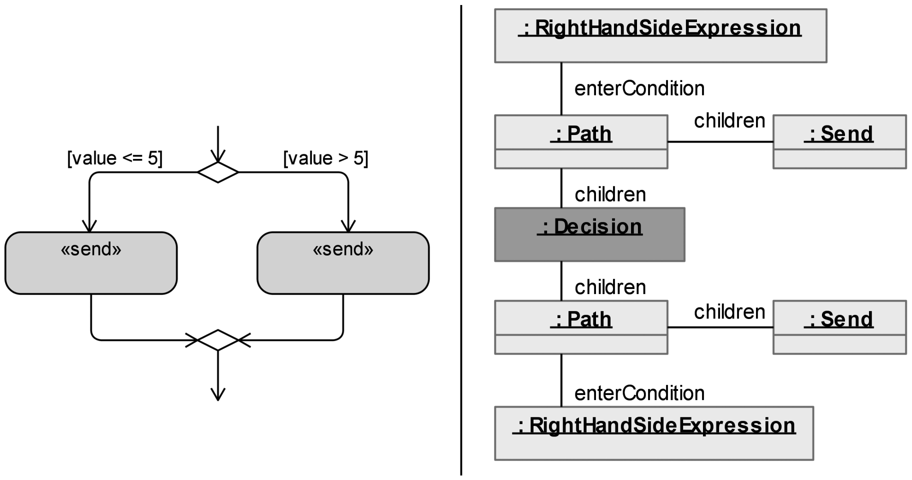


Figure 6.7: UML2SMM: Branch

Single Action Partition

The last partition type is the *Single Action Partition*, which consists of only one UML action to which, in most cases, a UML4SOA stereotype is attached — `CallOperationAction` (send, send&receive), `AcceptCallAction` (receive), `ReplyAction` (reply), `OpaqueAction` (compensate, compensateAll, data), `RaiseExceptionAction` and `ActivityFinalNode`. For each of these actions, the corresponding SMM equivalent is created and added to the current SMM composite element.

The most complex of these actions are the communicating actions, as they may have attached pins which must be converted to their own SMM model elements.

Transformation 9 (Communicating Actions) *Depending on the stereotype of the input action, an SMM `Send`, `Receive`, `SendAndReceive`, or `Reply` action is created. Each UML4SOA communication action has a pin which indicates the name of the partner of the communication. This name is resolved to an instance of an SMM `Service`, which is added as the `partner` of the communicating action.*

Next, the operation attached to the action is resolved — in case of send, send&receive and reply, the operation is given directly in the action; in case of a receive, the operation is attached to the `CallEvent` of the trigger. Either way, the operation is resolved to the corresponding SMM `ServiceOperation` and attached to the SMM action using the `operation` association.

Finally, send- and receive pins are handled. Send pins are converted to instances of SMM `SendParameter`, while receive pins are converted to instances of SMM `ReceiveParameter`; they are associated with the action using the `sndParameters` or `rcvParameters` association.

The expression specified in each send pin is converted to a ***`RightHandSide-Expression`****, which is attached as the* ***`source`*** *of the* ***`SendParameter`****; the expression specified in each receive pin is converted to a* ***`LeftHandSideExpression`*** *and attached as the target of the* ***`ReceiveParameter`****. Expressions are discussed further in the next section.*

An example for an UML4SOA receive operation which references the partner `provided` and the operation `providedOperation` from figures 6.2 and 6.4, respectively, is shown in figure 6.8. The example shows how the partner, the operation, and the (single) receive pin are attached to the SMM `Receive` element. Note that this example is not complete, as the `LeftHandSideExpression` is missing elements. Expressions are discussed in the next section.

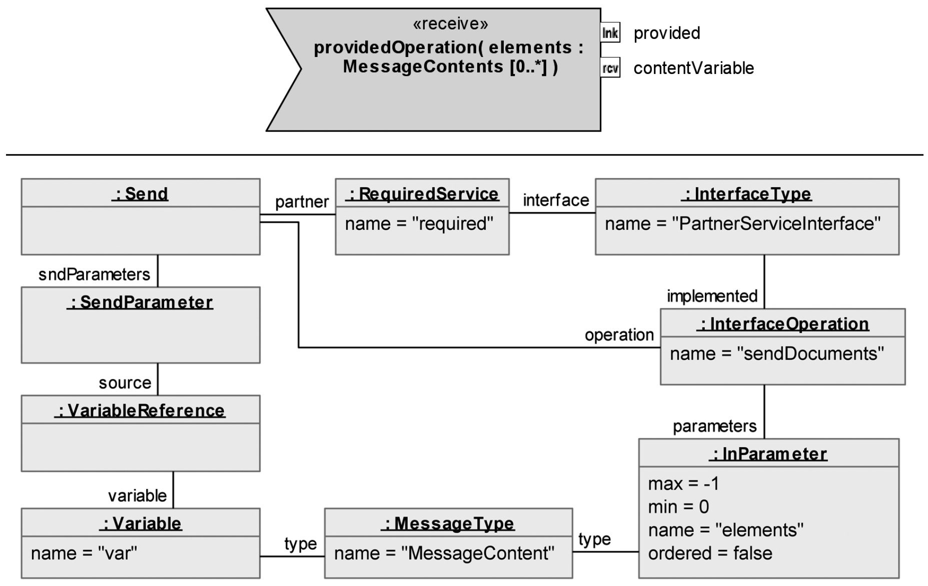


Figure 6.8: UML2SMM: Receive Action

The non-communicating actions of UML4SOA — data, compensate, compensateAll, and throw — also each have their corresponding SMM counterpart. Each is slightly different with regard to the attached information.

Transformation 10 (Data) *Each data action in the UML4SOA activity is converted to an SMM* ***`Data`*** *element. The body of a data action contains statements; these are converted to instances of SMM* ***`Statement`*** *and added to the* ***`statements`*** *association. Statements are discussed in the next section.*

Transformation 11 (Compensation Actions) *UML4SOA ≪Compensate≫ and ≪CompensateAll≫ actions are converted to SMM* ***`Compensate`*** *and* ***`Com-`***

`pensateAll` instances, respectively. In case of a compensate action, the name of the target service activity is given in the body of the action. The target activity is resolved, and the corresponding compensation handler (an SMM `ServiceActivity`) is attached to the SMM `Compensate` element by means of the `target` association.

Transformation 12 (RaiseExceptionAction) *An UML `RaiseExceptionAction` is converted to an SMM `Throw` instance. A `RaiseExceptionAction` has an attached value pin which carries the exception; the type of this exception is resolved to an SMM `ExceptionType` and added as the `exceptionType` of the `Throw`.*

Transformation 13 (Activity Final Node) *An UML activity final node aborts an activity; if this node is used somewhere within the activity this fact must be noted in the SMM. Therefore, an activity final node is transformed to an SMM `Terminate` action.*

This concludes the transformation of UML4SOA behaviour to the SMM. The next section deals with data handling.

6.2.4 Parsing and Converting Data Statements

Statements and expressions written in the UML4SOA data manipulation language may occur in two places in an UML4SOA model: In the body of ≪*Data*≫ actions, and in the body of pins (receive, send, and the value pins of `RaiseExceptionAction`). In the first case, statements are used, while the second consists of left-hand side expressions (receive pins) and right-hand side expressions (send and exception value pins).

Data statements and expressions are parsed according to the UML4SOA data manipulation grammar described in chapter 3 (section 3.2.4). During this process, instances of the corresponding meta-classes of the SMM discussed in chapter 4 are created and added to the SMM model of the service behaviour.

The parsing process deals with the usual intricacies of source code; for a detailed specification, see section 8.3. Here, the transformation steps for the three types of data manipulation elements occurring in UML4SOA are listed along with an example.

Transformation 14 (Statement) *A UML4SOA data manipulation statement may occur within the body of a ≪Data≫ action. There are two kinds of statements: Declarations, in which a variable is newly declared with a type, and assignments, in which a variable or property is assigned to. In the first case, an SMM `Declaration` instance is created. For the declared variable, a `Variable` is created and added to the service activity the ≪Data≫ action belongs to. To associate the declaration with its variable, a `VariableReference` instance is created and added as the `declaredVar` of the declaration. In the second case, an `Assignment` is created. An assignment has a left- and right-hand side, which are added as indicated in the next two steps.*

Transformation 15 (Left-Hand Side) *A left-hand-side specifies a container for data: In UML4SOA, this is either a variable, or the property of a variable if typed with a message type. Therefore, the transformation creates one of the concrete subclasses of* `LeftHandSideExpression`*:* `VariableReference` *or* `PropertyReference`*. While the former only links to a* `Variable`*, the latter links to both a property and its parent left-hand side. This enables not only referencing a property of a variable, but a property of a property as well. This structure is resolved during the transformation. Note that if a variable referenced in a left-hand side does not yet exist, it is created and added to the enclosing SMM* `ServiceActivity` *element.*

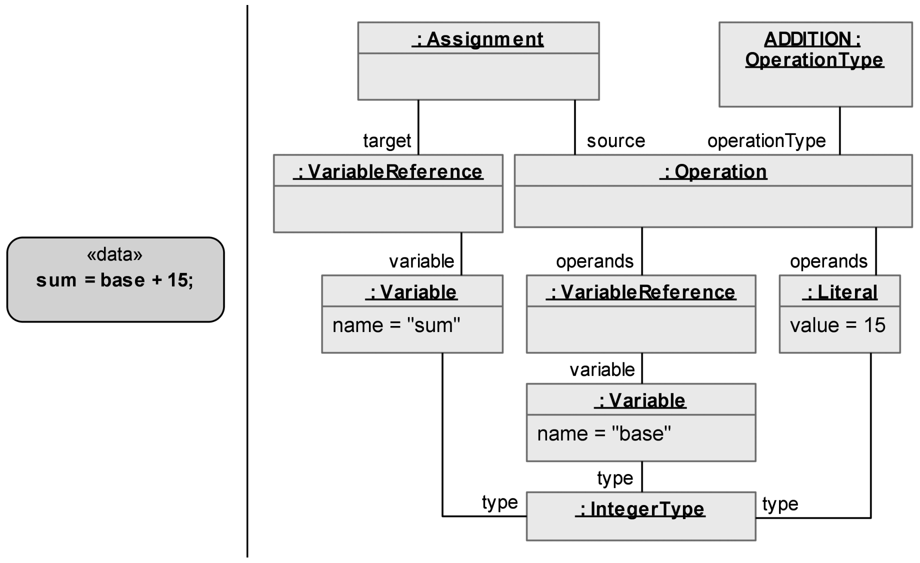


Figure 6.9: UML2SMM: Converting an UML4SOA Data Statement

Right-hand sides are used for several purposes in UML4SOA. The simplest one is specifying literals. A second purpose is performing calculations or string concatenations on literals, variables, and properties, or specifying conditional expressions through the use of operations such as *equals* or *greater than*. Finally, as the SMM `LeftHandSideExpression` is derived from `RightHandSide-Expression`, variable- or property references may also be used as right-hand sides.

Transformation 16 (Right-Hand Side) *Depending on the structure of the source code, the transformation either creates an SMM* `Operation` *with an appropriate* `OperationType`*, an SMM* `Literal` *with the appropriate* `value`*, or as discussed above one of the subclasses of* `LeftHandSideExpression`*. In the case of operations, the operands are handled recursively and added to the* `operands` *association.*

An example for the transformation of data statements is given in figure 6.9, which shows the conversion of an assignment inside a ≪*Data*≫ action to the SMM. The statement itself is converted to an SMM `Assignment`. The target of the assignment is an SMM `VariableReference` to the variable `sum`; the source is an SMM `Operation` with two operands: One is, again, a `VariableReference` to the variable `base`, the other is a literal with the value `15`. Both variables and the literal share a common type (`IntegerType`).

This concludes the discussion of the data statements of UML4SOA to the SMM.

6.2.4.1 Converting Protocols

As discussed in chapter 3, a UML4SOA protocol is specified as a UML protocol state machine. As for participant behaviour, we focus again on the UML4-SOA/Strict version of protocol definitions. The subset of UML protocol state machines we consider in UML4SOA/Strict consist of the following elements:

- *States.* The set of states includes the pseudostates start and end as well as normal states.
- *Transitions.* A transition is a directed edge between two states.
  - A transition may be annotated with a UML4SOA communication stereotype (one of ≪*Send*≫, ≪*Receive*≫, ≪*ReceiveReply*≫, or ≪*Reply*≫).
  - A transition may furthermore be annotated with the UML4SOA ≪*Optional*≫ stereotype.
  - If an annotation with a communication stereotype is given, the transition also must specify a trigger (either *SendOperationEvent* in case of ≪*Send*≫ and ≪*Reply*≫, or *ReceiveOperationEvent* in case of ≪*Receive*≫ or ≪*ReceiveReply*≫).

The SMM defines a very similar set of elements for protocols:

- *States.* A `ProtocolState` in the SMM contains a name. There is only one type of state; which state is the start state is defined in the `ServiceProtocol` element.
- *Transitions.* There are five transition types in the SMM; each transition may be declared optional with the `isOptional` flag. The first four types correspond to the UML4SOA stereotypes listed above; included are `SendingTransition`, `ReceivingTransition`, `ReplyingTransition`, and `ReceiveReplyingTransition`. The last type corresponds to a transition without a stereotype and is labeled `NoopTransition`.

From the two above lists, it follows that the mapping is rather straightforward. We can give three simple transition steps for states and transitions. First, we convert the protocol itself:

Transformation 17 (Service Protocol) *For each PrSM attached to an UML-4SOA port, an SMM `ServiceProtocol` element is created and attached to the respective `InterfaceType`.*

Second, we deal with protocol states:

Transformation 18 (Protocol States) *Each state in the UML4SOA PrSM is converted to an SMM `ProtocolState`, including pseudo initial and end states. The resulting converted state of the pseudo initial state is set as the start state of the SMM Protocol.*

Finally, we can convert the transitions.

Transformation 19 (Protocol Transitions) *Each transition in an UML4-SOA PrSM is converted to an SMM `ProtocolTransition` of the corresponding subtype. The `isOptional` flag is set to `true` if the PrSM transition has the ≪Optional≫ stereotype; otherwise, it is set to `false`. The subtype of `ProtocolTransition` is chosen as follows: `SendingTransition` is used for ≪Send≫ transitions, `ReceivingTransition` is used for ≪Receive≫ transitions, `ReplyingTransition` is used for ≪Reply≫ transitions, and `ReceiveReplyingTransition` is used for ≪ReceiveReply≫ transitions. Finally, if a transition does not have a communication stereotype attached, `NoopTransition` is used. The operation associated to the trigger of the PrSM transition, if exists, is resolved and attached as the `operation` of the SMM transition.*

This concludes the discussion of UML2SMM. In the following, instances of the SMM meta-model created by the UML2SMM transformation will be used as the source models for the SMM2WS and SMM2Java transformations.

6.3 SMM to Web Services

The first target for the transformation from UML4SOA is the Web Services family of standards [WCL+05], which consists of a number of interacting standards for defining Web Services. In particular, the transformation target chosen for the SMM behaviour is the Business Process Execution Language (BPEL) [OAS07]. BPEL is the de facto standard for Web Service orchestration and as such a citizen of the SOA world. Services and service communication, compensation, and event handling are first-class concepts in BPEL; thus, the programming style of UML4SOA, the SMM, and BPEL match in most cases. Still, there are some areas in which technical constructs are necessary to correctly map UML4SOA elements to BPEL. These will be detailed below.

Converting a complete SMM participant to the Web Service standards family requires not only the behavioural description in BPEL, but additional artefacts for describing the service interfaces, types, wire formats, addressing, and data manipulation. Thus, the SMM2WS transformation creates artefacts in BPEL, WSDL, XML Schema, XPath, and SOAP. The relevant standards from the Web Service family have been described in section 2. The transformation is implemented as a model-to-model transformation, thus, a meta-model for each of these languages is required; these are provided by the Eclipse Web Tools Platform project [Ecl10j] (WSDL, SOAP), the Eclipse Modelling Tools [Ecl10c] (XML Schema) and the Eclipse BPEL project [Ecl10g] (BPEL). XPath fragments are generated on-the-fly without a meta-model. Finally, WS-Addressing elements are not required at design time but will be used at runtime by the BPEL engine.

Each of the meta-models discussed above includes serialisation support; thus, the instances of the meta-models can be used to generate code in the actual target language such as BPEL or WSDL. Note that there is no standard graphical or non-XML textual notation for these meta-models. The following sections will thus use the serialised target code of the corresponding instances as examples.

The SMM2WS transformation creates one BPEL process for each of the behaviours of a participant. The BPEL process uses a set of WSDL files which describe both the services provided by the BPEL process and those provided by partners; the WSDL files in turn are based on the types declared in an XML Schema. The next section describes the static part of the transformation (WSDL and XSD), followed by the behavioural part (BPEL).

6.3.1 Transforming the Static Part

As in the transformation from UML4SOA to the SMM, the static part consists of two closely related elements: First, the definition of partners, partner operations, and their parameters; second, the message types available for communication. The transformation of partners and partner-related artefacts to WSDL is discussed in the next section; the conversion of the message types of the SMM to XML Schema is discussed in section 6.3.1.2.

6.3.1.1 Generating WSDL

As introduced in chapter 2, the functional description of a SOA participant is expressed using the Web Service Description Language (WSDL). The structure of a WSDL definition consists of an abstract part and a concrete part; the first includes types, messages, and port types, while the second contains bindings, services, and ports. This structure will be filled in the following transformation steps.

The SMM2WS transformation starts by generating a WSDL definition for each provided and required service of an SMM participant:

Transformation 1 (Services) *For each SMM `Service`, a WSDL `Definitions` element is created. The name of the definition is set to the partner name and a sensible name space is chosen and specified. In the `Types` section, the XML Schema of the process is referenced (see step 4).*

An example for the basic WSDL structure generated is shown in listing 6.1.

The second transformation step deals with the remaining abstract parts of the WSDL definition, i.e. port types and messages.

Both service- and request ports are transformed using the same algorithm. First, the main `PortType` of the WSDL definition is created. It captures the the operations invoked on the partner, and thus, includes WSDL versions of all operations *implemented* in the SMM `ServiceInterface`. Secondly, if the service interface also contains *used* operations, another port type with these operations is created: This is a *callback* port type, used to invoke operations on a communication counterpart of the current partner. WSDL services referencing the port type with the implemented operations of *provided services* are later *provided* by the generated BPEL process for the behaviours of the participant, while WSDL services referencing the port type with the implemented operations of *required services* are implemented by external partners and *invoked* by the BPEL process. The callback services are used in the complementary way.

Each SMM `ServiceOperation` is converted to a WSDL `Operation` inside of a `PortType`. A WSDL operation, however, may only contain one input message and (if necessary) one output message, whereas SMM operations may include several in- and output parameters. Thus, a maximum of two messages are created per operation: One containing all input parameters as WSDL `Parts`; and one using the output parameters as `Parts`.

Transformation 2 (Port Types, Operations, and Messages) *One `Port-Type` is generated for the partner with the partner name. For each implemented operation, a WSDL `Operation` is created along with input- and output `Mes-sages` containing the `Parts` corresponding to the operation parameters. If the partner has used operations, a callback port type is generated in the same way; to distinguish the port types, the callback port type is suffixed with "_callback".*

An example of the conversion of a service to an abstract WSDL definition is shown in figure 6.10 and listing 6.2, respectively. XML namespace definitions have been left out for clarity; `xs` refers to XML Schema and `this` to the current WSDL definition.

The third transformation step deals with the concrete part of a WSDL definition, which consists of bindings, services, and ports. A binding associates wire transportation formats to a port type, its operations, and input/output messages: A serialisation format for message transport is chosen, and an action name is associated with operations. In the case of UML2SMM, the `Bindings` generated always use SOAP RPC/Literal HTTP transport, and the SOAP Action is set to the original name of the operation.

Second, `Service` and `Port` elements are created. As indicated above, a maximum of two services are generated; one for the operations provided by the partner and one for the callbacks. Each service contains one port which references the appropriate binding, and which includes a target address where the service will be deployed (as part of tool support).

Transformation 3 (Bindings, Ports, and Services) *For each SMM `Port-Type` generated using the last step, a `Binding` element is created. The binding*

```
<wsdl:definitions name="partnerName"
        targetNamespace="http://www.mdd4soa.eu/generated/
                        partner/partnerName/">
  <wsdl:types>
    <xs:schema>
      <xs:import namespace="http://www.mdd4soa.eu/generated/ParticipantName/"
        schemaLocation="Participant.xsd"/>
    </xs:schema>
  </wsdl:types>
  <!-- more code here -->
</wsdl:definitions>
```

Listing 6.1: SMM2WS: Basic WSDL Structure

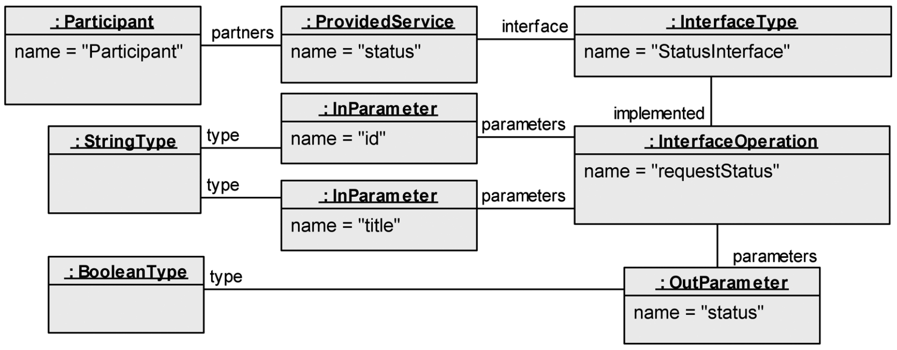


Figure 6.10: SMM2WS: Services, Operations, and Messages (SMM)

```
  <wsdl:message name="requestStatus_inMessage">
    <wsdl:part name="id" type="xs:string"/>
    <wsdl:part name="title" type="xs:string"/>
  </wsdl:message>
  <wsdl:message name="requestStatus_outMessage">
    <wsdl:part name="status" type="xs:boolean"/>
  </wsdl:message>

  <wsdl:portType name="status">
    <wsdl:operation name="requestStatus">
      <wsdl:input message="this:requestStatus_inMessage"
              name="requestStatus_input"/>
      <wsdl:output message="this:requestStatus_outMessage"
              name="requestStatus_output"/>
    </wsdl:operation>
  </wsdl:portType>
```

Listing 6.2: SMM2WS: Services, Operations, and Messages (WSDL)

is set to `RPC` *style and* `HTTP` *transfer. All operations of the port types are referenced and their name is set as the SOAP action. Each input or output is set to* `LITERAL` *style. Secondly, one* `Service` *element is created for each port type. A* `Port` *is added to the service, referencing the corresponding binding; finally, a (preliminary) SOAP address is added to the port.*

Listing 6.3 shows the concrete counterpart for the abstract WSDL section in listing 6.2. A binding has been created for the port type, referencing the chosen SOAP transport protocol and serialisation format; lastly, a service is available with a port referencing the binding.

```
<wsdl:binding name="status_binding" type="this:status">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="requestStatus">
      <soap:operation soapAction="requestStatus"/>
      <wsdl:input name="requestStatus_input">
        <soap:body namespace="http://www.mdd4soa.eu/generated/Participant/"
              use="literal"/>
      </wsdl:input>
      <wsdl:output name="requestStatus_output">
        <soap:body namespace="http://www.mdd4soa.eu/generated/Participant/"
              use="literal"/>
      </wsdl:output>
    </wsdl:operation>
</wsdl:binding>

<wsdl:service name="status">
  <wsdl:port binding="this:status_binding" name="status">
    <soap:address location="**DUMMY**"/>
  </wsdl:port>
</wsdl:service>
```

Listing 6.3: SMM2WS: Bindings, Ports, and Services (WSDL)

Finally, the WSDL definition needs to be enhanced with BPEL-specific extension elements, ensuring that the generated BPEL process is able to integrate with the WSDL definition. This requires the generation of partner link types, roles, properties, and property aliases; a discussion of these elements can be found in section 6.3.2.

6.3.1.2 Generating XSD

XML Schema specifications provide the ability to define the structure of XML documents. Since the standard way of communication between Web Services is using XML-based messages, XML Schema is used to define the structure of these messages as a whole and of the message parts which carry application data. The SMM2WS transformation creates XML Schema *complex types* with appropriate inner *elements* corresponding to the SMM `MessageType` and `MessageProperty` classes, respectively. XML Schema elements may contain a `minOccurs` and `maxOccurs` attributes, which match the semantics of the `min` and `max` SMM attributes.

Unfortunately, there is no useful native support for unordered sets in XML Schema: first, due to the fact that an XML element is a serialised tree whose elements are ordered in the document, second, as the `xsd:all` construct imposes some severe restrictions on the modeller, rendering it impracticable (for example, elements within an `xsd:all` may not be tagged with a `maxOccurs` value other than 1) (cf. also [Jam02]). For this reason, and due to the fact that an ordered list can always be viewed as unordered, no attempt is made to enforce unordered sets in the generated XML Schema types.

Each message type of the SMM is converted to an appropriate XML Schema type and is include in one XML Schema, which is referenced by the generated WSDL and BPEL files. This is captured in the next transformation step.

Transformation 4 (Message Types) *For each message type, an XML Schema `complexType` definition is created; an XML Schema `sequence` element is added as the first child. For each message property, an XML Schema `element` is created. If the property has a primitive type, the appropriate XML Schema primitive type is referenced. Otherwise, another complex type is created (with this step) and referenced. If a property is multi-valued, the `minOccurs` and `maxOccurs` attributes are set as appropriate.*

An example for the conversion of an SMM message type to XML Schema is shown in figure 6.11 and listing 6.4, respectively.

This concludes the transformation of XML Schema and WSDL artefacts. In the following, the behavioural part of the SMM model is converted to BPEL.

```
<xs:complexType name="Message">
  <xs:sequence>
    <xs:element minOccurs="0" maxOccurs="unbounded" name="contents"
          type="this:MessageContent"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="MessageContent">
  <xs:sequence>
    <xs:element minOccurs="0" name="content" type="xs:integer"/>
  </xs:sequence>
</xs:complexType>
```

Listing 6.4: SMM2WS: Message Types (XML Schema)

6.3.2 Transforming the Behaviour

The behavioural part of the SMM2WS transformation is concerned with generating one BPEL process for each behaviour of the transformed participant,

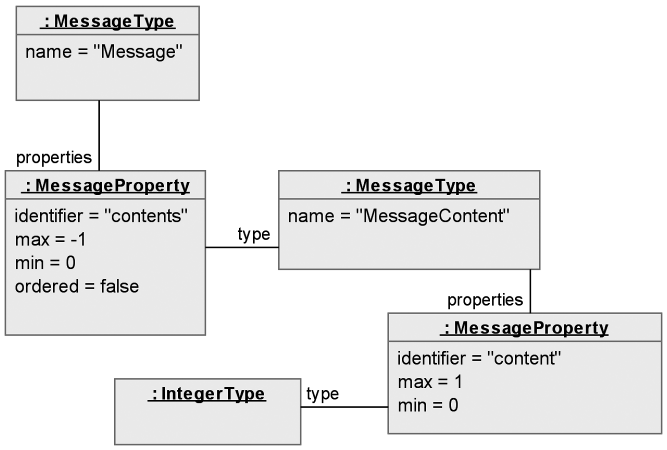


Figure 6.11: SMM2WS: Message Types (SMM)

which additionally necessitates some additions to the WSDL files by means of extension elements.

Transformation 5 (Behaviours) *For each SMM root `ServiceActivity`, a BPEL `Process` is created. The name of the definition is set to the behaviour name; a sensible name space is chosen and specified.*

```
<bpel:process name="BehaviourName"
   targetNamespace="http://www.mdd4soa.eu/generated/process/BehaviourName/">

  <!-- code here -->

</bpel:process>
```

Listing 6.5: SMM2WS: Basic BPEL Structure

An example for a basic BPEL process generated is shown in listing 6.5.

The structure of the SMM model and the layout of a BPEL process are similar: Both are based on nested structuring activities (service activities in the SMM, scopes in BPEL) and communication actions (send, receive, reply, and send&receive in the SMM, invoke, receive and reply in BPEL). Furthermore, both support the concept of exception, event, and compensation handlers and the invocation of the latter by specialised compensate actions. Finally, the SMM data manipulation language can be converted to a combination of BPEL assign statements and XPath expressions for addressing parts of a variable.

However, there are three areas in which the transformation needs to generate additional technical constructs:

- *Message-based communication.* Communication actions (`Invoke`, `Receive`, and `Reply`) in BPEL are based on sending or receiving single messages as defined in the WSDL operation specification. An SMM `ServiceOperation`, on the other hand, is based on in and out parameters with standard or custom data types. Thus, these parameters need to be placed in a message-typed variable to be sent before each `Invoke` and `Reply`, and need to be extracted from a message-typed variable after each `Receive`. This is done by means of a BPEL `Assign` action.
- *Instance matching.* During runtime, multiple instances of a BPEL process may be active. To be able to match incoming calls to the right instance, the middleware needs a handle for each process for matching messages to instances. In the absence of object identities, BPEL defines the concept of *correlation.* Thus, the transformation needs to generate correlation artefacts for this mechanism.
- *Interrupting Receives.* The SMM defines the concept of interrupting receives which are able to interrupt a service activity. As BPEL does not include such a concept, interrupting receives need to be simulated by using an specialised exception which interrupts a scope but is caught on the outside, such that the process may continue normally.

In the following, the SMM2WS transformation is discussed in three steps. The first step deals with the declaration of partners in both the BPEL process and the corresponding WSDL definitions, which is a requirement for communication with the outside world. The second step discusses generation of the actual behaviour inside the BPEL process, which includes service activity nesting, communication actions, handlers, and interrupting receives. Finally, the last step discusses correlation.

6.3.2.1 Defining Partners

For offering and requesting services, BPEL uses the notion of *partner links* and *partner link types.* A partner link type defines up to two *roles* which can be played by the BPEL process or an external service. Each role corresponds to a WSDL port type. Through the definition of a partner link based on the partner link type, the roles are assigned to the BPEL process itself (*my role*) or a partner (*partner role*).

The transformation of the static part of an SMM `Participant` has already created up to two port types for a service: One port type with the operations *implemented* by the service, and (possibly) another one with the operation *used* by the service. Depending on the type of the service, the roles corresponding to the generated port types are assigned as shown in table 6.1.

To illustrate this concept, the following list shows two common occurrences of provided and required services:

Partner Type	Operations	Assigned to
Service	Implemented	BPEL process (my role)
Service	Used	Remote Service (partner role)
Request	Implemented	Remote Service (partner role)
Request	Used	BPEL process (my role)

Table 6.1: Mapping of SMM Service Operations to Partner Roles

- *Service offered by the participant.* For an SMM participant with a provided service only implementing operations, one port type and one partner link type with one role are generated; the role is played by the BPEL process.
- *Service offered by a partner.* For an SMM participant with a required service only implementing operations, one port type and one partner link type with one role are generated, but the role is played by an external service.

While partner link definitions and role assignments are part of a BPEL process definition, partner link type and role definitions are part of the corresponding WSDL file. The following transformations step addresses both.

Transformation 6 (Partner Link Types and Partners) *For each SMM `Service`, a partner link type in the WSDL file of the service is created. If the service contains implemented operations, a role with the name of the partner and the corresponding port type is created. If the service contains used operations, an additional role with the name of the partner suffixed by "_callback" and the corresponding port type is created. In the BPEL process, the corresponding partner link based on the partner link type is created; the roles are set as shown in table 6.1.*

An example for the partner link types and partner links generated for the SMM participant shown in figure 6.10 is given in listing 6.6.

```
<!-- WSDL -->
<plnk:partnerLinkType name="status">
        <plnk:role name="status" portType="this:status"/>
</plnk:partnerLinkType>

<!-- BPEL -->
<bpel:partnerLink myRole="status" name="status"
            partnerLinkType="status:status"/>
```

Listing 6.6: SMM2WS: Partner Link Types and Partner Links (WSDL/BPEL)

6.3.2.2 Converting the Behaviour

As the nested structure of the SMM and BPEL is similar, a depth-first recursive algorithm is used in the conversion, starting with the root service activity. This is captured in the following transformation step:

Transformation 7 (Service Activity) *A service activity is converted to a BPEL `Scope` with the same name. The variables attached to the service activity are converted to BPEL variables with the corresponding XSD type and added to the scope. A BPEL `Sequence` is created inside the scope to ensure sequential behaviour. All handlers of the service activity are converted to BPEL handlers (step 8). Interrupting receives are handled as shown in step 9. Finally, the children of the service activity are handled one-by-one; for each child, the corresponding transformation step (steps 13 to 19) is executed.*

The transformation of a service activity to a scope yields the code shown in listing 6.7.

```
<bpel:scope name="ThesisManager">
   <bpel:variables>
         <bpel:variable name="id" type="xsd:string"/>
         <!-- more variables -->
   </bpel:variables>
   <bpel:sequence>
     <!-- code -->
   </bpel:sequence>
</bpel:scope>
```

Listing 6.7: SMM2WS: Scope with Initial Sequence (BPEL)

The next step in the conversion of a service activity is the transformation of handlers. For each SMM handler, a corresponding BPEL construct exists. An instance of this construct is created by the transformation according to the following considerations:

Compensation Handlers. The corresponding BPEL element for an SMM `CompensationHandler` is a `CompensationHandler` which shares the same semantics.

Event Handlers. The corresponding BPEL element for an SMM `EventHandler` is an `OnEvent` element inside a `EventHandlers` group. The `OnEvent` element is actually a specialised receive. The first receive of the SMM `EventHandler` is converted to the `OnEvent` element (for the conversion of receives, see step 14).

Exception Handlers. The corresponding BPEL element for an SMM `ExceptionHandler` is a `Catch` activity inside `FaultHandlers` group, which again shares the same semantics as the UML4SOA version. An SMM event handler references an exception type and variable name. In BPEL, a `faultName` is used to identify an exception. The SMM exception type name is used to generate this name.

Conversion of handlers is captured in the next transformation step.

Transformation 8 (Handlers) *Each SMM handler is converted to its BPEL counterpart as discussed above, and attached to the corresponding BPEL scope. An inner scope with an empty `Sequence` is generated for each handler. The children specified in the handler are then transformed one-by-one by the appropriate steps.*

The last construct attached to an SMM `ServiceActivity` are interrupting receives. An interrupting receive of an SMM service activity is active during the time the activity is active; if the receive is triggered, the activity along with all of its children is aborted and execution continues after the activity. There is no native BPEL counterpart for interrupting receives, thus, an interrupting receive is converted to a BPEL event handler with an `OnEvent` element for receiving the message, a `throw` element for interrupting the scope to which the event handler is attached, and an empty `Catch` attached to the scope to catch the fault.

Transformation 9 (Interrupting Receive) *For each interrupting receive of a service activity, a BPEL `EventHandler` is created. The receive is converted to an `OnEvent` element in the handler; in the sequence of the handler, a `throw` with a specialised fault name is added. Furthermore, a `FaultHandler` is attached to the scope which catches the same fault. The fault handler otherwise remains empty.*

An example for a scope with an interrupting receive is shown in listing 6.8.

Before the transformation steps for the children of a service activity can be discussed, another important aspect must be handled: The communication actions, the data handling statement, and the structured actions of the SMM are all based on the data access and manipulation statements from the SMM data package. Thus, the right- and left-hand side expressions of the SMM need to be converted to appropriate XPath expressions and placed in BPEL `Assign` activities. This is discussed next.

In general, data handling is performed in BPEL by using `Assign` activities. Each assign activity may have number of `Copy` children, which copy data from a source to a target, where the source may include operation calls (as usual on a right-hand side). BPEL `Copy` elements distinguish between different kinds of copy operations; different notations are used for the copy target for WSDL message parts, standard BPEL variables, and inner elements of variables with complex XML Schema types. The same holds true for the copy source, which distinguishes between the same elements as above and additionally allows a

```
<bpel:scope name="Main">
  <bpel:onEvent
    <!-- receive specification -->  >
    <bpel:scope name="Interrupting_Receive">
      <bpel:sequence>
        <!-- assign for receive -->
        <bpel:throw faultName="types:InterruptionFault"/>
      </bpel:sequence>
    </bpel:scope>
  </bpel:onEvent>

  <bpel:faultHandlers>
    <bpel:catch faultName="types:InterruptionFault">
      <!-- empty -->
    </bpel:catch>
  </bpel:faultHandlers>
</bpel:scope>
```

Listing 6.8: SMM2WS: Scope with Interrupting Receive (BPEL)

literal element for predefined XML fragments. The following steps do not distinguish between the different notations; in each case `to` and `from` clauses with appropriate attributes and expressions are generated.

Transformation 10 (Data Statements) *There are two kinds of statements in the SMM: Declarations and assignments. The former is used to initialise a variable, i.e. declaring it in the current scope; the latter assigns a value to a variable or property. In the first case, step 7 has already created a BPEL variable corresponding to the declaration, as a variable is always attached to its service activity. The initialisation step consists of creating the inner structure of the variable; using a BPEL `Assign` and a `Copy` child, a literal XML fragment corresponding to the XML Schema type is copied into the variable. In the case of an assignment, an SMM `RightHandSideExpression` is assigned to a `LeftHandSideExpression`. This is again handled with a BPEL `Assign` and `Copy` element; details are discussed in the next two steps.*

Transformation 11 (Data Storage using Left-Hand Sides) *A left-hand side in the SMM is either a `VariableReference` or a `PropertyReference`. Left-hand sides are used to denote a place to store data — as targets of assignments, or as receive parameters in receive operations. In both cases, the `from` element of a BPEL `Copy` activity is used to indicate the variable. Additionally, in the case of a property reference, a query path is attached to the variable to indicate the assigned subelement.*

Transformation 12 (Data Retrieval using Right-Hand Sides) *Finally, a right-hand side in the SMM is a `Literal`, an `Operation`, or a `LeftHandSideExpression`. In each case, the converted element is placed in the `from`* *element of a BPEL `Copy` activity. The first is converted to an XML literal. Note that this is not always a trivial task, as the XML Schema needs to parsed and a literal XML fragment constructed according to the requirements of the schema. In the second case, the SMM `OperationType` must be converted to an XPath operation; however, the XPath operations match quite naturally to the SMM operation types. Finally, `VariableReference` and `PropertyReference` right-hand sides are converted as in step 11.*

An example for the conversion of a data statement is shown in figure 6.12 and listing 6.9.

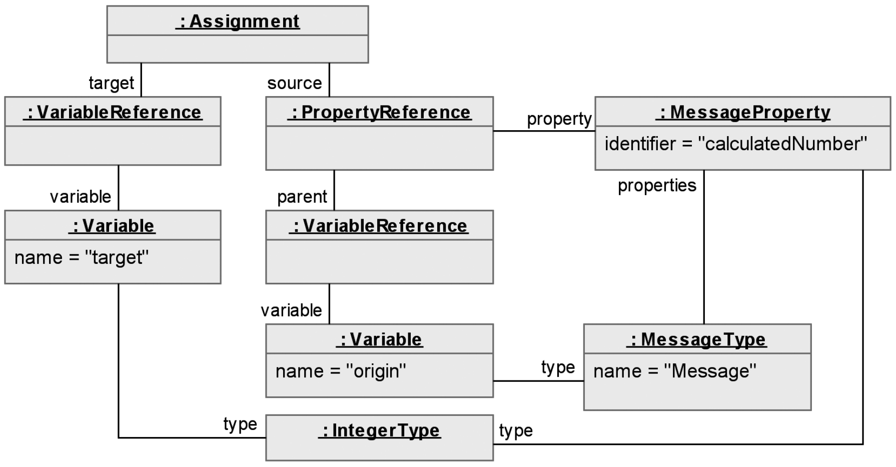


Figure 6.12: SMM2WS: Assignment (SMM)

```
<bpel:assign>
  <bpel:copy>
    <bpel:from>$target</bpel:from>
    <bpel:to>$origin/types:calculatedNumber</bpel:to>
  </bpel:copy>
</bpel:assign>
```

Listing 6.9: SMM2WS: Assignment (BPEL/XPath)

The children yet to be handled can be categorised into three groups. The first group consists of the SMM communication primitives (send, send&receive, receive, reply); the second group consists of primitive structuring actions (data, throw, compensate, and compensate all), and the third group consists of structured elements (loop, parallel, and decision).

Communication Primitives

The SMM actions `Send` and `Send&Receive` each model a call from the participant to the outside. A `Send` might have input parameters which are SMM right-hand sides; a `Send&Receive` may additionally have return parameters which are SMM left-hand sides. However, as mentioned in section 6.3.1, BPEL operations may only use WSDL messages as in- and output. Thus, an appropriate message must be constructed before the send operation out of the right-hand sides of the SMM send parameters. The same holds true for the returning message: The parts of the received message are extracted and copied into the variables specified as the SMM receive parameters.

Transformation 13 (Send and Send&Receive) *Both `Send` and `Send&Receive` SMM actions are converted to a BPEL `Invoke` activity, which references the partner link corresponding to the referenced SMM partner and the WSDL operation corresponding to the referenced SMM operation. For the data to be sent, a BPEL variable with the appropriate WSDL message type is created and attached to the current scope. Before the `Invoke`, an `Assign` activity is inserted into the BPEL code, which copies the right-hand sides specified in the `sndParameters` into the appropriate message parts of the generated message variable. In case of a `Send&Receive`, an additional `Assign` activity is inserted after the `Invoke`, which extracts the message parts from the received message and copies them to the appropriate variables specified as the left-hand sides in the `rcvParameters`. Again, the message itself is stored in a BPEL variable attached to the current scope.*

An example for transforming a `Send` from the SMM to BPEL and XPath is shown in figure 6.13 and listing 6.10.

A `Receive` action is converted to a BPEL `Receive` activity. As in the second part of a `Send&Receive`, the message-typed variable resulting from the operation is split into its parts in a subsequent `Assign`.

Transformation 14 (Receive) *An SMM `Receive` action is converted to a BPEL `Receive` activity which references the BPEL partner link corresponding to the SMM partner and the WSDL operation corresponding to the SMM operation attached to the receive. To be able to store the receive message, a BPEL variable with the appropriate WSDL message type is created and attached to the current scope. An `Assign` activity is inserted after the `Receive`, which extracts the message parts from the received message and copies them to the appropriate variables specified as the left-hand sides in the `rcvParameters` of the `receive`.*

Finally, a reply operation is converted in the same way as a send.

Transformation 15 (Reply) *An SMM `Reply` action is converted to a BPEL `Reply` activity, which references the partner link corresponding to the referenced SMM partner and the WSDL operation corresponding to the referenced SMM*

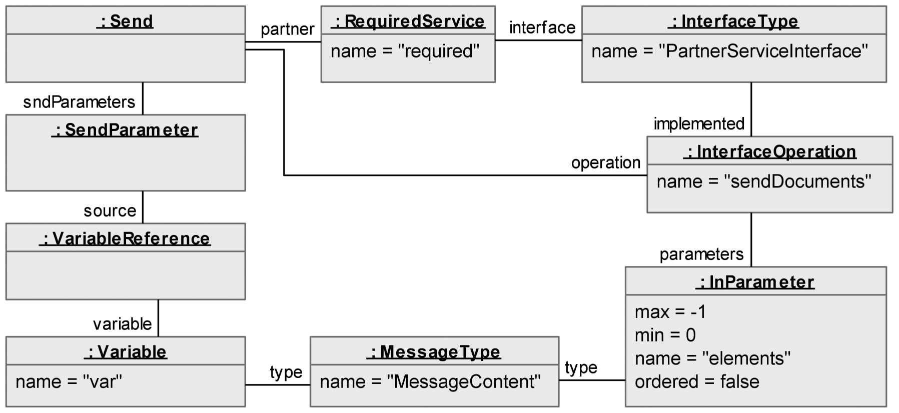


Figure 6.13: SMM2WS: Send (SMM)

```
<bpel:assign>
  <bpel:copy>
    <bpel:from>$var</bpel:from>
    <bpel:to part="doc" variable="sendDocument_send"/>
  </bpel:copy>
</bpel:assign>
<bpel:invoke
  operation="sendDocument" inputVariable="sendDocument_send"
  partnerLink="required" portType="required:required">
```

Listing 6.10: SMM2WS: Send (BPEL/XPath)

operation. To ensure that the reply is connected to its receive, a randomly generated `messageExchange` identifier is added to both BPEL elements. For the data to be sent, a BPEL variable with the appropriate WSDL message type is created and attached to the current scope. Before the `Invoke`, an `Assign` activity is inserted into the BPEL code which copies the right-hand sides specified in the `sndParameters` into the appropriate message parts of the generated message.

Primitive Structuring Actions

The group of primitive structuring actions consist of `Throw`, `Data`, `Compensate`, and `CompensateAll`.

A `Throw` action raises an exception, which ends at least the current scope (and subscopes); possibly more until handled. In BPEL, this concept is called fault handling. Instead of exceptions, fault handling is based on (qualified) fault names. In the transformation, such names are generated based on the process namespace and the simple name of the exception type.

A `Data` action performs a set of data statements on the variables of the participant. The statements are translated as indicated in the corresponding transformation step with a BPEL `Assign` element.

Finally, both the `Compensate` and `CompensateAll` SMM actions invoke and wait for the compensation handler of the indicated, or all inner, scopes. The BPEL equivalent for the compensate action is the `CompensateScope` activity, which contains a link to the target scope to be compensated; the equivalent for the compensate all action is the `Compensate` element. The semantics are identical.

Transformation 16 (Primitive Structuring Actions) *For an SMM `Throw`, a BPEL `Throw` activity is created with a fault name as indicated above. For an SMM `DataHandling` action, a BPEL `Assign` activity is generated; the individual statements are converted as shown in step 10. For `Compensate` and `CompensateAll`, the BPEL activities `CompensateScope` and `Compensate` are generated; in the former case, the name of the target scope is attached to the activity.*

Structuring Elements

For each of the structured activities of the SMM, a corresponding BPEL activity exists: An SMM `Loop` is converted to a `RepeatUntil` activity; a `Parallel` is converted to a `Flow` with inner `Sequences`, and a `Decision` is converted to `If/ElseIf/Else` activities.

Transformation 17 (Loop) *An SMM `Loop` is converted to a BPEL `RepeatUntil` activity. The `leaveCondition` of the SMM loop is transformed to XPath using step 12, negated, and used as the condition for the `RepeatUntil` activity. A BPEL `Sequence` is added to the `RepeatUntil` statement to ensure ordering, and the children of the SMM `Loop` are each converted and added to the sequence as usual. Note that if no `leaveCondition` exists, the condition is set to `false`; in this case, only an interrupting receive or throw can lead the process out of the loop.*

An example for a loop transformation is shown in figure 6.14 and listing 6.11.

Transformation 18 (Decision) *An SMM `Decision` is converted to BPEL `If/ElseIf/Else` activities. The `enterCondition` of the first path encountered is transformed to an XPath expression using transformation step 12 (right-hand sides) and used as the condition for the first BPEL `If` statement. For all subsequent paths, an `ElseIf` statement is created with the appropriate `enterCondition` and attached to the preceding `If`. Note that only one of the conditions may be missing; in this case, a final else statement is employed for the corresponding path. Inside each `If` or `ElseIf` activity, a BPEL sequence is created; the children of the corresponding SMM path are added to these sequences as usual.*

Transformation 19 (Parallel) *An SMM `Parallel` is converted to a BPEL `Flow` activity. For each path in the parallel activity, a new BPEL `Sequence` is created and added to the flow. The children of the corresponding SMM path are added to these sequences as usual.*

The discussion of the transformation of structuring activities concludes the main transformation from the Service Meta-Model (SMM) to the BPEL programming language.

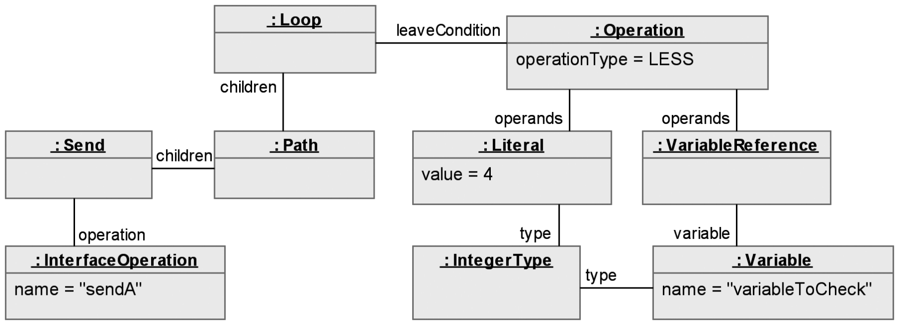


Figure 6.14: SMM2WS: Loop (SMM)

```
<bpel:repeatUntil>
  <bpel:sequence>
    <bpel:invoke> ... </bpel:invoke>
  </bpel:sequence>
  <bpel:condition>$variableToCheck == 4</bpel:condition>
</bpel:repeatUntil>
```

Listing 6.11: SMM2WS: Loop (BPEL)

6.3.2.3 Correlation

In the comparison of UML4SOA/Open and UML4SOA/Strict, the concept of instance matching has already been mentioned (cf. section 3.2.6): As a participant — including its associated behaviours — only describes a single instance of a service execution, a middleware needs to take care of starting, handling, and stopping these instances. In the case of BPEL, this function is performed by a BPEL container such as Apache ODE [Apa10b], the ActiveBPEL engine [Act10], or the Oracle BPEL Process Manager [Ora10].

If multiple instances of a BPEL process are running at the same time, incoming calls from partners must be routed to the correct instance. In UML4SOA (and, as the next transformation shows, in Java), this problem is solved by the object-oriented principle of object identity — once a port (or partner object) is instantiated, the identity of the port associated to the behaviour takes care of appropriately routing calls. BPEL, on the other hand, defines the concept of *correlation* to achieve instance matching, which is not based on the object identity of ports or messages; instead, certain elements in the message payload are chosen as *correlation ids* which identify the concrete instance a call is targeted at.

Considering the eUniversity case study already used in the previous chapter, correlation is only required for partners with callbacks, i.e. student and tutor. A good candidate for a correlation element is the *thesis id*, as it unambigously identify the process instance (since one BPEL process instance handles one thesis).

As correlation IDs are only required for certain transformation targets such as BPEL, correlation is not part of the general definition of UML4SOA as described in chapter 3. Rather, the transformation offers the option of adding correlation to the process, requiring the developer to select a set of correlation elements in the generated messages and automatically creating the required artefacts.

Correlation in BPEL is based on four concepts: Properties, property aliases, correlation sets, and message correlation. The first two are extensions to WSDL, while the latter two are used in the BPEL process.

Transformation 20 (Properties and Property Aliases) *For each set of correlation elements chosen by the developer, a `Property` is generated in a new WSDL definition imported by all partner WSDL definitions. For each correlation element, a `PropertyAlias` is defined in the corresponding partner WSDL definition, binding the correlation element to the property.*

Transformation 21 (Correlation Sets and Message Correlation) *For each set of correlation elements chosen by the developer, a `CorrelationSet` is created in the BPEL process referencing the corresponding generated WSDL property. In each communication with the selected partners, a message `Correlation` element is added, referencing the corresponding set. For the first of these communications in the process, the `initiate` attribute is set to true.*

This concludes the discussion of the SMM2WS transformation.

6.3.3 Semantics of SMM and BPEL

The aim of the transformation from the SMM to BPEL — with surrounding artefacts — has been enabling developers to ultimately convert from UML4SOA to code, while keeping to the semantics defined for the SMM and UML4SOA laid down in chapter 5.

The semantics defined for the SMM matches the generated BPEL code in many cases; in particular, the nesting construct, parallel, loop, and decision handling, as well as compensation handling is the same. However, there is no one-to-one mapping to the MIO semantics, which lies in the mismatch of the UML4SOA and BPEL specifications. This hinders an exact mapping from the SMM to BPEL, which affects the following SMM elements:

- *Communication Actions.* The BPEL specification leaves the exact handling of receiving and sending actions open — i.e., it is not clear for example in a one-way `invoke` whether the message must in fact be received by the partner (with an acknowledgement): Each engine may implement its own strategy. By contrast, the UML4SOA semantics is very strict about these matters.

- *Interrupting Edges.* An interrupting receive, in the SMM, effectively aborts an activity and all event handlers (although they are allowed to finish) and then moves on to the next element. There is no concept for interrupting receives in BPEL. Thus, to implement an interrupting receive, an exception is thrown, caught, and then the following actions are allowed to continue. As this is a multi-step approach, there might be other parallel actions interjecting during this process which is not possible in the SMM.

- *Event Handlers.* The BPEL specification allows event handlers to run in parallel; not only to others but also to themselves. This is not allowed in UML4SOA; here, a second event handler of the same type must wait for the first. This problem is somewhat alleviated by the fact that the protocol has to allow this too; nevertheless, the execution semantics is different.

Finally, it has been shown that different BPEL engines in fact implement different semantics [LPT09], which further complicates an exact mapping. Requiring UML4SOA-like semantics in BPEL, however, would lead to a new implementation of a BPEL engine. We have therefore opted to address this level of preciseness in the Java transformation in the next section.

6.3.4 Case Study Example

The eUniversity case study already presented in the previous chapters has been converted using the SMM2WS transformation. The generated Web Service artefacts for the eUniversity case study are comprised of one BPEL process (as there is only one activity attached to the `ThesisManager` participant), five WSDL definitions (Blackboard, Examination Office, Graduation Service, Student, and Tutor), and one XML Schema which contains four complex types (Thesis, Student, Grade, and Document). Of the five WSDL definitions, three contain only one service (Blackboard, Examination Office, and Graduation Service), while the other two contain an additional callback service (Student and Tutor).

A graphical representation of the static structure of the converted eUniversity case study is shown in figure 6.15. The behavioural part of the case study (i.e. the BPEL process `ThesisManager`) consists of 421 lines of code and is thus not shown here. The full implementation may be downloaded from the links provided in section 8.3.

6.4 SMM to Java

The second target language and paradigm chosen for transformation is the object-oriented programming language Java. As an object-oriented language, Java follows a method-invocation, object-based programming paradigm which is different from the communication-based workflow structure of UML4SOA and SMM service behaviours. There are two options for dealing with this mismatch.

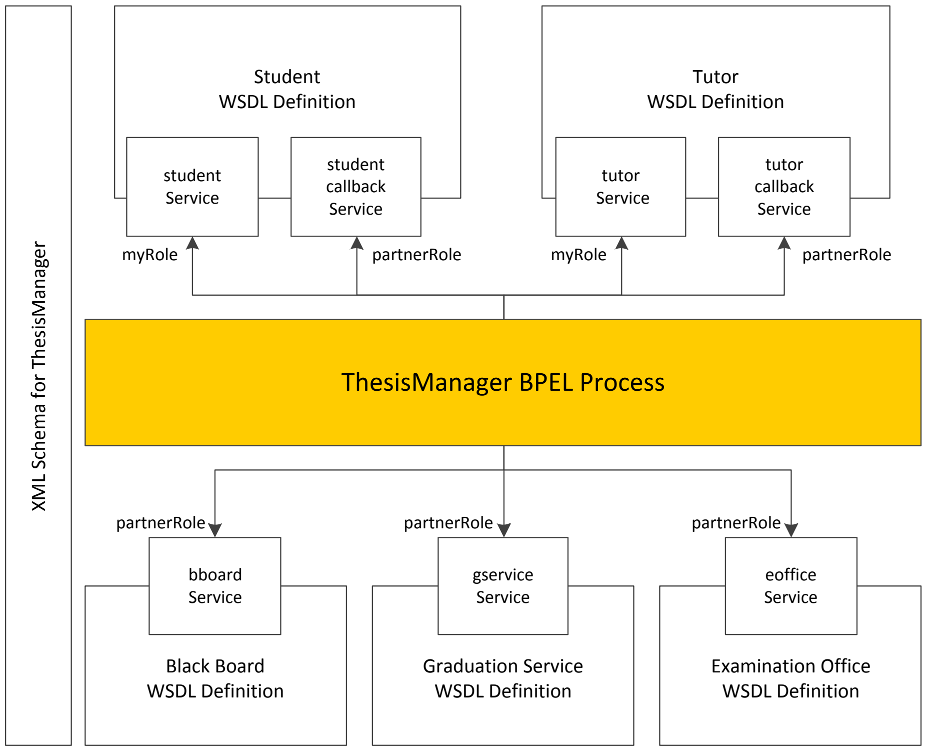


Figure 6.15: eUniversity Case Study: Static Part in BPEL

The first option places emphasis on the source, i.e. SMM and the UML4-SOA meta-model and semantics. This, of course, has the advantage of truly capturing the ideas behind UML4SOA. On the other hand, these concepts do not lend themselves naturally to Java concepts and thus, the transformation needs to create code which differs from the usual Java programming style.

The other possibility is placing emphasis on the target, i.e. trying to generate code with naturally fits the Java paradigm, sacrificing an exact mapping from UML4SOA for native, straightforward Java code.

This choice depends on one's view of MDD. The former view is that of MDD as a compilation step. In this view, the only interesting — and edited — artefact is the model, while the rest is machine code not to be read by humans. The second view is that of MDD as a helping hand in programming: Models are seen as a first step and the generated code is seen as a skeleton, later to be completed in the target programming language.

While both views have their application areas, this section takes the first approach, as the aim here is an exploration of SOA concepts such as communication between services, compensation, and event handling — which includes the question of how to faithfully capture these concepts in a traditional programming language. For the same reason, no specific SOA libraries for Java, like for example Apache Axis [Apa10a], are used — instead, the converted code only uses functionality available within the core Java API.

As usual, the SMM2Java transformation is implemented as a model-to-model transformation between two EMF meta-models. The first is the SMM meta-model which has been introduced in chapter 4; the second is the MoDisco [Ecl10e] meta-model, which is a reflection of the Java language as defined in the Java Language Specification [GJSB05]. The meta-model consists of 132 classifiers and 213 structural features. Instances of the MoDisco model can later be serialised to actual code; this is discussed in the tools section. For readability, the following discussion uses plain Java code in the transformation examples.

6.4.1 Transforming the Static Part

As in the previous sections, transforming the static part of the system is a prerequisite to the behaviour. As the static part of the SMM is essentially object-oriented, the mapping of the corresponding SMM constructs to Java code does not require a significant add-on to what is provided by the language.

The SMM2Java transformation starts with the root element of the SMM — a participant, which may have several behaviours. Each behaviour is represented in Java as a class. As the static aspects of a participant apply to all behaviours, all generated classes share the corresponding features as well. A participant has provided and required services, each referencing `implemented` and `used` operations. In Java, an operation which is invokable on the behaviour must be declared in an interface *implemented* by the (behaviour) classes, while operations invoked on partners must be declared in an interface *used* by the classes. In general, a maximum of two Java interfaces are created for each service — one for the required, and one for the provided operations. Table 6.2 shows how operations are mapped to interfaces, and how these interfaces are used by participants.

Partner Type	**Operation**	**Mapping to Java**
Service	Implemented	Interface implemented by participant
Service	Used	Interface referenced by participant
Request	Implemented	Interface referenced to participant
Request	Used	Interface implemented by participant

Table 6.2: Mapping of SMM Services to Java Interfaces

The transformation of participants and their services is captured in the following first transformation step.

Transformation 1 (Participants and Services) *For each root service activity of an SMM participant, a Java class is created. The* **`partners`** *associated to the participant are transformed as listed in table 6.2: For each partner, a* *maximum of two interfaces is created, one with a "_required" suffix, one with a "_provided" suffix. Provided interfaces are implemented by the behaviour classes, while a field is created for required ones; the field is set via constructor injection. In each generated interface, the corresponding operations are defined as methods; the parameter and return types are transformed as shown in step 2. For provided interfaces, method stubs are generated in all behavioural classes.*

An example for the conversion of a participant with a service is shown in figure 6.16 and listing 6.12. The first figure shows the SMM model, while the listing shows the converted Java code.

Message and exception types, in both UML and the SMM, already follow the object-oriented idea of classes which encapsulate their relationships and attributes. Thus, the message types can be converted as-is to Java classes. Exception types are treated likewise, but must extend the Java `Exception` class and provide certain constructors to be useful.

An interesting point to note is that both the UML and the SMM handle multiplicity as being attached to an element instead of a type — i.e. the meta-classes `Variable`, `Parameter`, and `MessageProperty` all have a lower and upper bound as well as an indication whether they are ordered. This idea is similar to the UML `MultiplicityElement` meta-class and also matches the `minOccurs` and `maxOccurs` attributes in XML Schema (section 6.3).

In Java, however, while there is built-in support for arrays with an upper bound, these are always ordered; furthermore, modern Java programs usually do not rely on arrays but on parameterised version of the `Set` and `List` interfaces provided as part of the Java collections API — i.e., the notion of multiplicity is not attached to an element but to its type. Additionally, ensuring a maximum occurrence of a property is usually done by explicit logic rather than constraints on the type (which would require a specific set or list implementation). In our transformation, all maximum occurrences other than one are thus treated as being unbounded. If the property is ordered, a parameterised list is generated; otherwise, a parameterised set.

Message types are converted as shown in the next transformation step. An example for this transformation step is shown in figure 6.17 and listing 6.13, respectively.

Transformation 2 (Message- and Exception Types) *For each message and exception type, a Java class is created. For each property, a field is created. If the property is defined with a primitive SMM type, the corresponding Java type is used; in case of a `MessageType`, step 2 is invoked for the target type and the result is referenced in the declaration. In case the property is multi-valued and ordered, a parameterised Java `ArrayList` is created; if unordered, a `HashSet`. Furthermore, getters and setters are created for all fields. Finally, if the type is an exception type, an extends clause for `java.lang.Exception` is added along with the default constructors.*

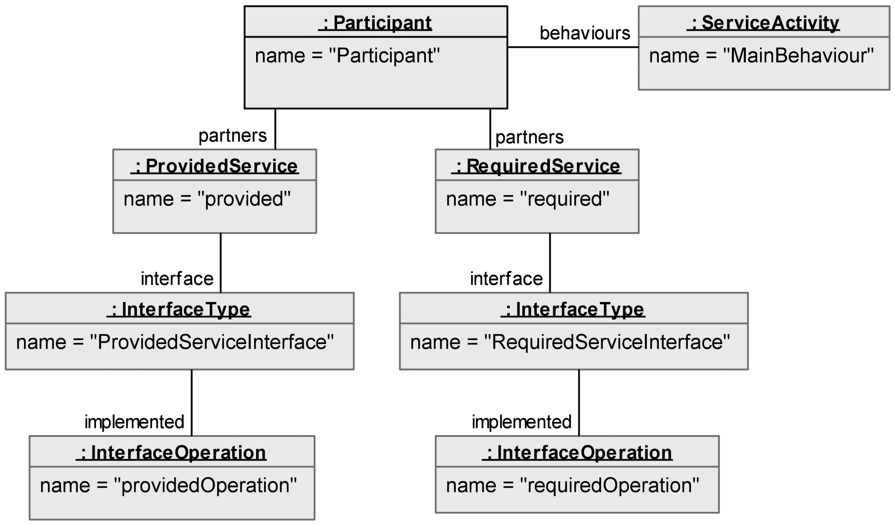


Figure 6.16: SMM2Java: Participants and Interfaces (SMM)

```
interface ProvidedServiceInterface_Provided {
   void providedOperation();
}

interface RequiredServiceInterface_Required {
  void requiredOperation();
}

class MainBehaviour implements ProvidedServiceInterface_Provided {

 private RequiredServiceInterface_Required partner_required;

 public Participant(RequiredServiceInterface_Required required) {
    partner_required= required;
 }

 void providedOperation() {
        // code here
 }

}
```

Listing 6.12: SMM2Java: Participants and Interfaces (Java)

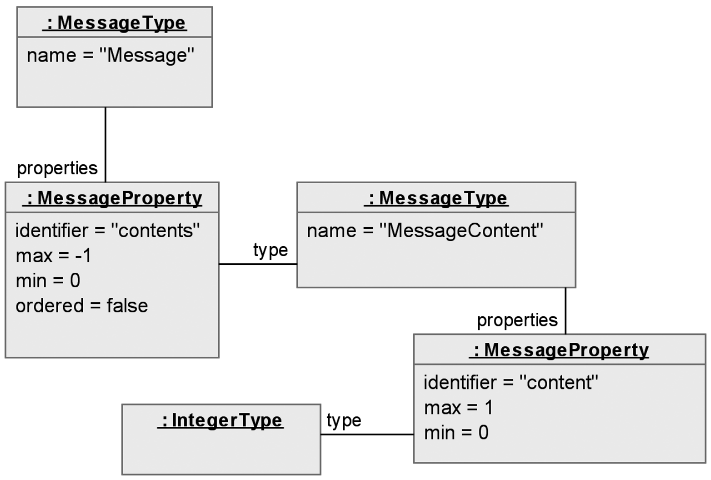


Figure 6.17: SMM2Java: Message- and Exception Types (SMM)

```
class Message {

  private HashSet<MessageContent> contents;

  public void setProperties(HashSet<MessageContents> contents) {
    this.contents= contents;
  }

  public HashSet<MessageContent> contents() {
    return contents;
  }
}

class MessageContents {

   int content;

   // etc

}
```

Listing 6.13: SMM2Java: Message- and Exception Types (Java)

6.4.2 Transforming the Behaviour

The behavioural part of the SMM2Java transformation is concerned with implementing the bodies of the Java classes created in step 1 for each behaviour of the transformed participant.

As mentioned above, the SOA-inspired ideas behind UML4SOA differ in some key points from the standard semantics of Java. The following list identifies requirements to be considered when transforming SMM instances to Java code.

- Firstly, as children of the networked world, services and service orchestrations are inherently parallel: Using event handlers, parallel blocks, or interrupting receives in an UML4SOA service description easily leads to a situation in which more than one part of the behaviour is active at the same time. This problem necessitates the first requirement for implementing UML4SOA behaviours in Java: Individual parts of the behaviour — service activities, handlers, parallel blocks, and interrupting receives — must execute in *parallel*. In the following, these parts are named *execution scopes*.
- A second issue arises when considering the structure of UML4SOA activities and the structure of Java programs. UML4SOA and SMM models are well-nested: Each execution scope has a parent and children, each handler and interrupting receive belongs to one service activity, and variable access as well as exception handling is based on nested elements. Thus, another requirement on the SMM2Java transformation is a concept for declaring and exploiting the *parent/child relationship of execution scopes*.

Placing child execution scopes under the direct control of their parents has several benefits in the implementation. First, when a service activity completes or is interrupted, it can stop its still-running children (like event handlers), waiting for them to complete as required by UML4SOA. Second, an exception thrown in a certain scope is handled by interrupting the scope, setting an exception flag, and then interrupting the parent who can deal with the situation appropriately (if it has an exception handler, handle the exception — otherwise, propagate it). Finally, the scopes for interrupting receives can stop themselves and indicate to their parent to interrupt, which the parent can handle as usual (by stopping all other children and then itself). The following list contains the required operations on an execution scope.

- *Starting scopes.* Execution scopes must be able to start their child scopes — whether they are handlers, interrupting receives, or plain nested activities. Handlers and interrupting receives are started as soon as a service activity starts, while other service activities are only enabled as soon as they are encountered in the sequential flow.
- *Waiting for scopes.* Execution scopes must be able to wait for the regular end of a scope. Note that the regular end might also be the start (in case of an event handler).

- *Forcing scope termination.* If an execution scope must be aborted for some reason, it needs to be able to force its children to close. Note that in the case of interrupting receives, the parent scope is forced to close.

- *Propagating exceptions.* If an exception is thrown in an execution scope without an attached handler, it needs to be stored and propagated to the parent scope.

- *Storing and retrieving variables.* Variables are attached to scopes, and can be retrieved from them. As the structure of scopes is composite, retrieving a variable traverses the tree to the root scope until a variable is found.

Finally, a third requirement needs to be taken into consideration, which affects the implementation of communication actions in Java:

- The third issue concerns the implementation of ≪*Receive*≫ (and as shown later, ≪*Reply*≫) actions in Java. Quite naturally, the ability to receive an operation invocation is modelled as a method, which follows from the implementation of the corresponding provided interface as discussed in the last section. However, while (public) methods in a Java class are at all times visible and invokable from the outside, UML4SOA ≪*Receive*≫ actions are usually not enabled during the complete execution of a behaviour. Instead, they are only invokable in certain states: This is both indicated by their placement in the activity workflow and in the corresponding protocol. This means that a ≪*Receive*≫ is only possible once the workflow has reached the action, and can only continue if a call comes in, i.e. invoker and invoked must wait for one another until they can perform the ≪*Receive*≫ together. Attempting to capture this idea in Java leads to a third requirement for the transformation: methods provided for receives must contain a mechanism which *synchronises* the invoker with the currently active execution scope of the service implementation.

There is another, more subtle problem associated with implementing incoming receive actions as methods in Java: As noted above, the ≪*Receive*≫ action is performed by both invoker and invoked together. However, this is also the only step which is performed in unison; after the receive action has completed, each party continues independently. In particular, any subsequent steps the UML4SOA behaviour executes are *not* part of the execution of the receive (unless the receive is linked to a later ≪*Reply*≫, see below). This concept is in contrast to the default semantics of Java methods: When implementing a method to handle a ≪*Receive*≫, the statements following the receive must not be executed as part of the method body, but after the method has returned. Thus, the Java implementation may not contain subsequent actions of a receive inside the receive method body, but must execute them as part of the execution scopes discussed above.

The inverse problem occurs for ≪*Reply*≫ actions, which returns an answer to a previous receive. Before a method which implements a receive may return, the action in-between receive and reply must have been executed. Thus, a reply requires another synchronisation between the invoker and the currently active execution scope. Note that this is not a problem for send and send&receive operations: The responsibility for synchronisation lies with the call receiver, which in this case is the partner.

In the following, Java constructs for addressing these requirements are discussed, before moving on to the actual transformation steps.

6.4.2.1 Mapping UML4SOA to Java

The three issues mentioned above can be addressed with two constructs from the Java language and API. The first construct are *threads*; the second are *barriers*.

The need to perform several task in parallel is addressed in SMM2Java by employing Java *threads* (with accompanying runnables). The transformation creates one runnable for each execution scope (service activities, handlers, interrupting receives, and parallel blocks) in the input behaviour. Although not strictly necessary in all cases, unifying the representation of scopes greatly simplifies the transformation and increases readability — each execution scope can be started, stopped, and interrupted in a standard way.

The transformation provides the pre-implemented class `ServiceRunnable` which is subclassed by all generated runnables. `ServiceRunnable` contains methods for starting a thread based on the runnable, waiting for a thread to complete, force-ending a thread, and storing exceptions. To be able to wait for the runnable to complete, it contains a flag — `isCloseable` — which indicates whether the current state of the runnable allows the runnable to be closed (in most cases, this state is the same as the thread `isAlive` flag, but there are exceptions). The methods provided are the following:

- The `start()` method creates a new thread based on the `ServiceRunnable`. The state of the service runnable is set to active, and the thread is started. The start method contains one parameter — the parent execution scope, which is stored for use in `interruptParent()` (see below).
- The `waitForEnd()` method waits for the runnable to reach its end and then returns. This is a graceful stop of the runnable.
- The `forceEnd()` method interrupts the runnable, forcing it to shut down (after cleaning up).
- The `interruptParent()` method calls `forceEnd()` on the parent given in `start()`.
- The `setExcception()` method notes the fact that an exception has been thrown in this thread. More information on exceptions is given below.
- The `addVariable()` method adds a named variable with a certain value to the scope.

- The `getVariable()` returns the value of a named variable from the scope or, recursively, the parent of the scope.

The `run()` method of each `ServiceRunnable` is implemented as part of the transformation and follows the natural sequential flow described in the SMM input. The pattern for creating the `run()` method is shown in listing 6.14 (an enumeration `ExecutionScope` listing the available scopes, and a map `runnables` containing the scope runnables are, for now, assumed to be present).

The example shows the `run()` method of a service activity scope which has one event handler and one inner scope. The event handler is started before the actual scope actions are executed; before the scope finishes, the event handler is allowed to finish.

```
void run() {
  try {
        runnables.get(ExecutionScopes.EventHandler).start(this);
        // ...
        runnables.get(ExecutionScopes.InnerScope).start(this);
        runnables.get(ExecutionScopes.InnerScope).waitForEnd();
        // ...
        runnables.get(ExecutionScopes.EventHandler).waitForEnd();
  } catch (InterruptedException e) {
        runnables.get(ExecutionScopes.InnerScope).forceEnd();
        runnables.get(ExecutionScopes.EventHandler).waitForEndEnd();
  }
  setCloseable(true);
```

Listing 6.14: SMM2Java: Execution Scope Runnable Example Code

The inner scope is started at the appropriate place; as this is a sequential invocation, the behaviour waits for the inner scope to complete. In case of an interruption, the inner scope is forced to finish. Note that an event handler is always *allowed* to finish when running, as this is a requirement of UML4SOA and is needed when interruptions by interrupting receives occur. Finally, as the last statement, a flag is set in the runnable to indicate that the thread may be closed at any time now.

Synchronisation of an external call with the currently executed scope — as indicated in the third requirement listed above — is essentially a thread synchronisation problem: In each case, two threads — the invoking thread executing the provided method and the thread running the execution scope which contains the corresponding receive (and possibly reply) — need to wait for one another, performing the operation together.

The Java concurrency API contains the concept of *barriers* which address this issue. A barrier consists of an object on which a specified number of threads can synchronise — as soon as all threads arrive at the barrier, a *barrier runnable* is executed, after which each thread continues on its own. This concept is shown in figure 6.18. In SMM2Java, all barriers are used to synchronise only two threads: An external invoker thread and one of the threads of the participant.

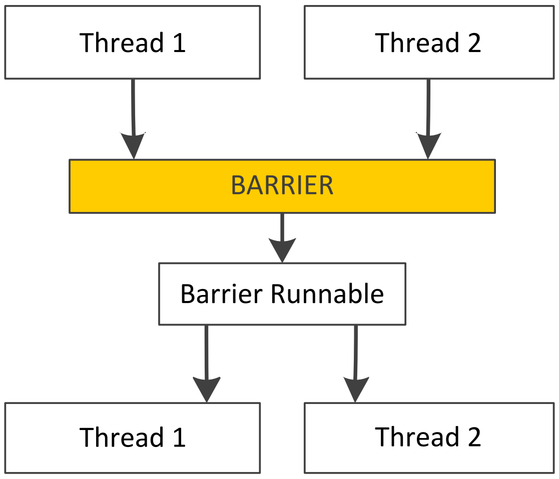


Figure 6.18: SMM2Java: Barrier Concept

The barrier runnable is used to change the state of the behaviour in an atomic fashion. As such, it is responsible for handling the data associated with a receive or reply — in the first case, storing new data, in the second retrieving it. Similar to the implementation of the execution of scope runnables, the SMM2Java transformation provides a common superclass for barrier runnables, `BarrierRunnable`, which contains two methods for storing and retrieving the arguments of the call (`setArguments()` and `getArguments()`, respectively). Again, the `run()` method is implemented as required during the transformation.

A receive or reply call thus requires the generation of code in three places in the participant class: In the provided method, in the execution scope where the receive was originally placed, and in the barrier runnable. The generation pattern is shown in listing 6.15 (note that an enumeration `Barriers` listing the available barriers and two maps `barrierRunnables` and `barriers` are assumed to present. They will be introduced in the corresponding transformation step).

The upper part of the listing shows the implemented method from the provided interface, in which the given argument is set on the runnable, and the code then proceeds to wait for the active behaviour to arrive at the barrier. The middle part shows a part of the active behaviour, which also waits for the barrier. Together, they then perform the code in the lower part, which is the run method of the barrier runnable.

With the concepts of service runnables, barriers, and barrier runnables in place, the remaining issues in transforming the behaviour of an UML4SOA participant to Java can be handled as part of the transformation description listed in the next section.

```
void externalCall(String parameter) {
  synchronized(lock_partner) {
        barrierRunnables.get(Barriers.startService).setArguments(parameter);
        barriers.get(Barriers.startService).await();
  }
}

void run() {
  //...
  barriers.get(Barriers.startService).await();
  //...
}

void run() {
  addVariable(ExecutionScopes.Main, "parameter", getArguments().get(0));
}
```

Listing 6.15: SMM2Java: Barrier Example Code

6.4.2.2 Transforming the Behaviour

Implementing the structure discussed above requires a multi-step approach to the transformation: First, the required anonymous inner classes and methods are created with their method bodies remaining empty. In a second step, the methods are implemented with the appropriate code. Note that the first set of empty methods — the implemented methods of the provided interfaces — has already been created in the static section.

The following steps are executed for each behaviour attached to the SMM participant. The transformation begins with the execution scopes.

Transformation 3 (Structure of Execution Scopes) *An enumeration `ExecutionScopes` is generated in the behaviour class. For each service activity, handler, interrupting receive, and path in a parallel element, an enumeration constant with the appropriate name is created. Furthermore, a hashmap (`ExecutionScopes` $\rightarrow$ `ServiceRunnable`) with the name `scopes` is created. In the constructor already created in the static part, an anonymous subclass of ServiceRunnable is created for each scope and added to the `scopes` map. The `run()` method is left empty.*

The second step concerns the barriers; the same initialisation is required as for the execution scopes.

Transformation 4 (Structure of Barriers) *An enumeration `Barriers` is generated in the behaviour class. For each receive and reply, an enumeration constant with the appropriate name is created. Furthermore, two hashmaps are created: `barriers` (`Barriers` $\rightarrow$ `CyclicBarrier`) and `barrierRunnables`*

(`Barriers` → `BarrierRunnable`) is created. In the constructor already created in the static part, an anonymous subclass of `BarrierRunnable` is created for each enumeration constant and added to the `barrierRunnables` map. The `run()` method is left empty. Furthermore, for each enumeration constant, a new instance of `CyclicBarrier` is created with the corresponding `BarrierRunnable` as a parameter. The barrier is set to wait for two threads, and is added to the `barriers` map.

The pattern generated by the last two transformation steps is shown in listing 6.16. In the upper part, the enumerations and maps are shown. In the lower part, the constructor already introduced in the static section is shown again. Here, the maps are initialised and the runnables and barriers created. Note that the run() methods are not yet implemented, and that the barriers are initialised to wait for two threads, and use the barrier runnable created for them.

The SMM2Java transformation uses the following transformation steps to convert one execution scope to Java, adding code to the `run()` method of the scope runnable, and — if the scope contains receive or reply actions — to the corresponding `run()` methods of the barrier runnables and the method bodies of the implemented methods of the provided interfaces.

Transformation 5 (Service Activity Behaviour) *A try/catch statement is added to the `run()` method of the service runnable to ensure that started child scopes can be closed. At the beginning of the try statement, event handlers are started with a call to `start()` of the corresponding service runnable. At the end of the try statement, `waitForEnd()` is invoked on the same runnables; in the catch statement, `forceEnd()` is invoked for all inner scopes except for event handlers, for which `waitForEnd()` is invoked. An example for this code has already been given in listing 6.14.*

The SMM children of the execution scope are handled one-by-one. For each child, the corresponding transformation step (see below) for the type of the child (for example, receive, inner service activity, or data) is executed.

A path in a parallel execution scope is similar to a service activity, except that it does not contain any handlers. Thus, the above transformation step can be used for the paths of parallel scopes as well. For event handlers, the following step is used:

Transformation 6 (Event Handler Behaviour) *In addition to the statements added in the step for service activities, a `while(true)` loop is added to the try statement, as event handlers may be executed more than once. An event handler is optional; i.e. while waiting for the initial receive, it can be cancelled. Thus, before the first receive statement, the state of the runnable is set to be `closeable`; after the receive statement, it is set to `not closeable` (the invoker must wait for the event handler to finish once it has been started).*

```
private enum ExecutionScopes { ScopeA, ... };

private HashMap<ExecutionScopes, ServiceRunnable> scopes;


private enum Barriers { ReceiveA, ... };

private HashMap<Barriers, BarrierRunnable> barrierRunnables;

private HashMap<Barriers, CyclicBarrier> barriers;


public BehaviourClass( ... ) {

        // field setup (see static part)

        scopes= new HashMap<ExecutionScopes, ServiceRunnable>();
        scopes.put(ExecutionScopes.ScopeA, new ServiceRunnable() {
                public void run() {
                        //...
                }
        });

        // more scopes

        barrierRunnables= new HashMap<Barriers, BarrierRunnable>();
        barrierRunnables.put(Barriers.ReceiveA, new BarrierRunnable() {
                public void run() {
                        //...
                }
        });

        // more barrier runnables

        barriers= new HashMap<Barriers, CyclicBarrier>();
        barriers.put(Barriers.ReceiveA, new CyclicBarrier(2,
                barrierRunnables.get(Barriers.ReceiveA));

        // more barriers
}
```

Listing 6.16: SMM2Java: Runnable Setup

Finally, interrupting receives of service activities have their own runnables, as they execute in parallel with their main scope, interrupting it if necessary. This is captured by the following step:

Transformation 7 (Interrupting Receive Behaviour) *An interrupting receive only consists of one statement — the receive. Nevertheless, the same steps as in the service activity step are executed. As an interrupting receive has the task of interrupting its parent when triggered, the last statement in the try block is an* `interruptParent()` *call.*

An interrupting receive is thus basically handled as a normal receive (see below) in its own execution scope. One more caveat is attached to an interrupting receive, however: The semantics states that a) event handlers are allowed to run even after an interrupting receive has occurred, and b) event handlers are no longer allowed after the next call is received. To ensure this behaviour, the actions immediately following an interrupting receive in the SMM must be prefetched and executed before the event handler is stopped. The following transformation step ensures this behaviour.

Transformation 8 (Prefetching Interrupting Receive Follow-Ups) *The communication actions immediately following an interrupting receive are moved from their original place to the beginning of the interruption-handling block of the execution scope in which the interrupting receive is handled. They thus take place before the event handlers are canceled, which is done within the same* `synchronized`*-block (discussed below) to ensure atomicity.*

The communication actions of the SMM, as well as the data statements and conditions used in structured elements may contain data expressions, which need to be transformed to Java. The following transformation steps deal with data statements.

Transformation 9 (Data Statements) *There are two kinds of statements in the SMM: Declarations and assignments. The former is used to initialise a variable, i.e. declaring it in the current scope; the latter assigns a value to a variable or property. In the first case, the method* `addVariable()` *is invoked on the current scope with the name of the variable and a newly created instance of the variable type. In the second case, an SMM* `RightHandSideExpression` *is assigned to a* `LeftHandSideExpression`*. This is discussed in the next two steps.*

Transformation 10 (Data Storage using Left-Hand Sides) *A left-hand side in the SMM is either a* `VariableReference` *or a* `PropertyReference`*. Left-hand sides are used to denote a place to store data — as targets of assignments, or as receive parameters in receive operations. There is a fundamental difference in Java between adding a new variable or setting a property. In the case of a* `VariableReference`*, the intention is adding, or replacing the value of the variable. Thus, the generated code is* `addVariable()` *with the name and* *value of the variable to set. In the case of a* ***PropertyReference****, a variable must already exist. Thus, the generated code invokes the setter for the property using* ***getVariable().setProperty()****, with the property name replaced appropriately.*

Transformation 11 (Data Retrieval using Right-Hand Sides) *Finally, a right-hand side in the SMM is a* ***Literal****, an* ***Operation****, or a* ***LeftHandSideExpression****. The first is converted to a Java literal as expected. In the second case, the SMM* ***OperationType*** *must be converted to Java. The types can be directly mapped to Java operators.*

It is important to note, however, that using a ***LeftHandSideExpression*** *as a right-hand side is a different use case to the previous transformation step: The aim of right-hand sides is to retrieve data, not to store it. Thus, a* ***VariableReference*** *in a right-hand-side is converted to a* ***getVariable()*** *statement instead of* ***addVariable****. Likewise, the generated code for* ***PropertyReference*** *invokes the getter for the property using* ***getVariable().getProperty()*** *instead of using a setter.*

An example for the conversion of data statements from the SMM to Java is shown in figure 6.19 and listing 6.17.

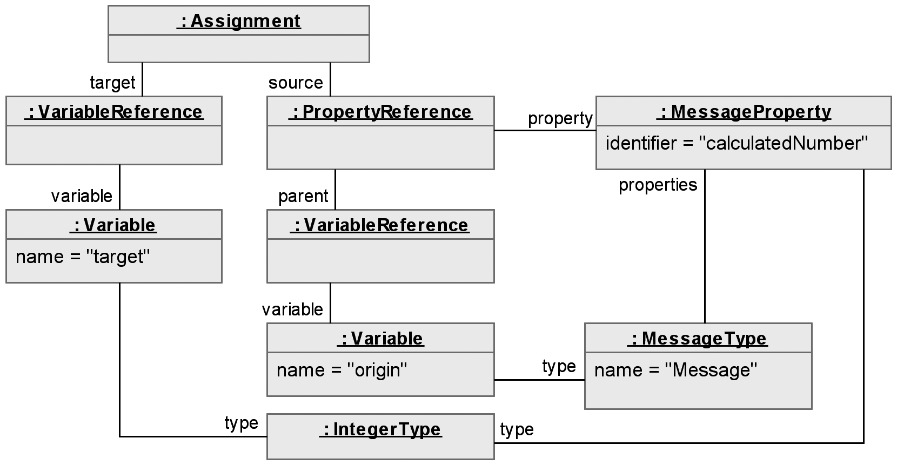


Figure 6.19: SMM2Java: Assignment (SMM)

```
setVariable("target", getVariable("origin").getCalculatedNumber());
```

Listing 6.17: SMM2Java: Assignment (Java)

The children yet to be handled can be grouped into three categories: communication primitives (send, send&receive, receive, reply), primitive structuring actions (throw, data, compensate, compensate all), and structured elements (service activity, decision, loop, and parallel).

Communication Primitives

There are four communication primitives to be handled: `Send`, `Send&Receive`, `Receive`, and `Reply`. Each of these actions references a certain partner. In order to ensure that only one action by a partner is active at the same time, all communication actions are enclosed in `synchronized`-blocks using a lock for the corresponding partner. The next transformation introduces these locks.

Transformation 12 (Communication Locks) *For each partner, a field is generated in the implementation. The field has the name of the partner prefixed with* *`'lock_'`*; *has an* *`Object`* *type and is directly initialised.*

The SMM actions `Send` and `Send&Receive` model a call from the participant to the outside. A send might have input parameters which are SMM right-hand sides; a send&receive may additionally have return parameters which are SMM left-hand sides; both can be converted as shown above. Thus, the transformation step only consists of generating the call:

Transformation 13 (Send and Send&Receive) *Both* *`Send`* *and* *`Send&Receive`* *SMM actions are converted to Java method calls on the field corresponding to the partner. Each parameter is converted as shown in the right-hand side transformation step. In case of a* *`Send&Receive`*, *the result of the method call is assigned to the appropriate variable or property as shown in the left-hand side transformation step. The method call is enclosed in a* *`synchronized`*-*block using the appropriate partner lock.*

An example for transforming a `Send` from the SMM to Java is shown in figure 6.20 and listing 6.18.

A `Receive` action requires the participant to wait for an incoming message. Handling a receive thus requires adding code for a) waiting for the barrier in the scope runnable, b) waiting for the barrier in the implemented service method, and c) performing the data assignment in the barrier runnable. An example of the converted code has already been given in listing 6.15 on page 243.

Transformation 14 (Receive) *In the* *`run()`* *method of the current execution scope, an* *`await`* *call for the receive barrier is placed. In the provided method for the receive, parameters of the method are stored in the barrier runnable using* *`setArguments()`*, *and the second* *`await`* *call for the barrier is added. Finally, the arguments are retrieved using* *`getArguments()`* *in the* *`run()`* *method of the barrier runnable, and stored as appropriate in a variable or property. The body of the provided method for the receive is enclosed in a* *`synchronized`*-*block using the appropriate partner lock.*

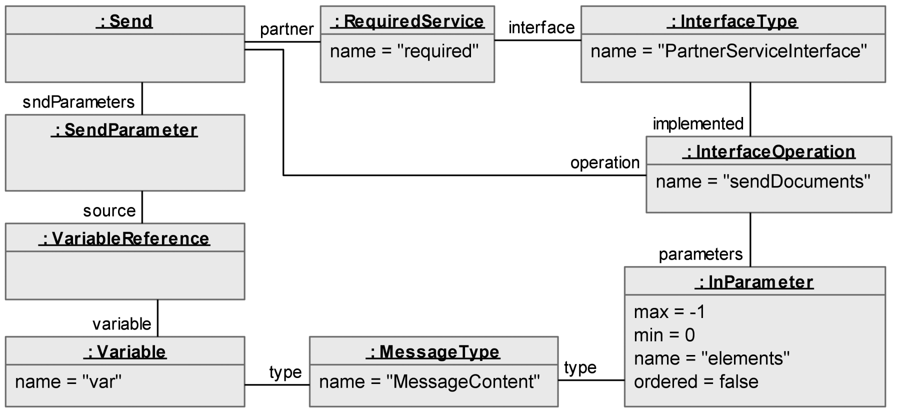


Figure 6.20: SMM2Java: Send (SMM)

```
synchronized(lock_required) {
  partner_required.sendDocument((Document) getVariable("var"));
}
```

Listing 6.18: SMM2Java: Send (Java)

A `Reply` action is used to answer a call received in a previous receive action. It requires an additional reply barrier: The scope runnable needs to wait for the reply barrier upon encountering the reply, and the implemented service method needs to wait for the reply barrier after the receive barrier has been passed. Thirdly, as in the case of receive, the reply barrier runnable needs to handle the data. The code therefore looks similar to the one given in listing 6.15 on page 243.

Transformation 15 (Reply) *In the `run()` method of the current execution scope, an `await` call for the reply barrier is placed. In the provided method for the receive associated with the reply, the second `await` method call is placed after the receiving code; after the `await` call, a return statement is added with the arguments retrieved by using `getArguments()` on the reply runnable. Finally, in the `run()` method of the reply barrier runnable, the return values are retrieved using `getVariable()` and stored in the runnable by using `setArguments()`.*

Primitive Structuring Actions

The group of primitive structuring actions consists of `Throw`, `DataHandling`, `Compensate`, and `CompensateAll`.

A `Throw` action raises an exception which ends at least the current scope (and sub-scopes) — possibly more, until handled. As indicated in the previous section, throwing standard Java exception is not possible due to the nested structure of the SMM inherited by the Java implementation. Therefore, the generated code notes the exception and interrupts itself. The code for invoking a service activity runnable is responsible for handling the exception, as detailed in the next paragraph.

A `DataHandling` action includes a set of data statements manipulating the variables of the participant. The statements are translated as indicated in transformation step 9.

Finally, both the `Compensate` and `CompensateAll` SMM actions invoke and wait for the compensation handler of the indicated, or all inner, scopes. Since SMM2Java generates one runnable for each handler, such a runnable exists and can directly be invoked in the same way as a nested activity.

Transformation 16 (Primitive Structuring Action) *For an SMM `Throw` element, a `setException()` call is placed and an `InterruptedException` is thrown. For an SMM `DataHandling` element, the individual statements are converted as shown in transformation step 9. For `Compensate` and `CompensateAll`, one or several compensation handler runnables are started using `start()` and allowed to finish with `waitForEnd()`.*

Structuring Elements

The structuring elements to be handled in this section are `ServiceActivity`, `Decision`, `Loop`, and `Parallel`.

The transformation of `ServiceActivity` and its inner components has been the subject of the discussion of most of the transformation steps above, starting with transformation step 3 (structure of execution scopes). What has not yet been fully discussed is the process of starting and ending the service runnable of a service activity, in particular in combination with exception handling.

Exception handling is done by starting an exception handler, which, as usual for execution scopes, is transformed to a runnable itself. However, a service activity might not have an exception handler attached although an exception is thrown. In this case, the exception is simply passed on to the next parent.

Transformation 17 (Invoking Service Activities) *A service activity is invoked by executing the `start()` method of the corresponding runnable. To wait for an activity, the `waitForEnd()` method is used. This method returns if the end of the runnable is reached, but also in case of an exception in the service activity. Thus, the method `getException()` is used after `waitForEnd()` to check for an exception. In case an exception is found, and a corresponding exception handler is attached to the current scope, this handler is executed, again by means of `start()` and `waitForEnd()`. In case an exception is found but no event handler is present, the exception is propagated with `setException()` and throwing an `InterruptedException` as shown in transformation step 16.*

An example for starting a service activity and passing on an exception is given in listing 6.19.

```
runnables.get(ExecutionScopes.InnerActivity).start(this);
runnables.get(ExecutionScopes.InnerActivity).waitForEnd();
if (runnables.get(ExecutionScopes.InnerActivity).getException() != null) {
        setException(runnables.get(ExecutionScopes.InnerActivity).getException());
        throw new InterruptedException();
}
```

Listing 6.19: SMM2Java: Handling Exceptions

An SMM `Decision` composite element models a conditional check; as a result, only one of several possible paths of elements are executed. Each path contains a condition which must be checked before entering the path. A decision can be transformed naturally to Java by using an `if-else` statement.

Transformation 18 (Decision) *A decision is converted to a Java `if-else` statement. The `enterCondition` of the first path encountered is transformed to a Java expression using transformation step 11 (right-hand sides) and used as the condition for the first `if` statement. For all subsequent paths, another `if` statement is used in the else branch of the preceding `if`. Note that one of the conditions may be missing; in this case, the final `else` statement is employed for the corresponding path. The children of the paths are handled one-by-one as usual and added to the body of the `if` statement generated for the path.*

An example for converting an SMM `Decision` to Java code is shown in figure 6.21 and listing 6.20, respectively. The SMM model contains two paths; one with an `enterCondition`, the other without. The send statements are not fully specified to keep the diagram readable.

```
if (((Integer) getVariable("variableToCheck")) < 4) {
  partner.sendA();
} else {
  partner.sendB();
}
```

Listing 6.20: SMM2Java: Decision (Java)

An SMM `Loop` composite element models recurring behaviour. The loop body is executed at least once; depending on an *exit condition*, the runnable may step out of the loop. In Java, the `do-while` loop may be used for modelling this behaviour; note that the exit condition must be negated to achieve the right semantics.

Transformation 19 (Loop) *A loop is converted to a Java `do-while` statement. The `leaveCondition` of the loop is transformed to a Java expression using transformation step 11 (right-hand sides) and used as the condition for the `while` statement. The children of the loop are added within the `do-while`*

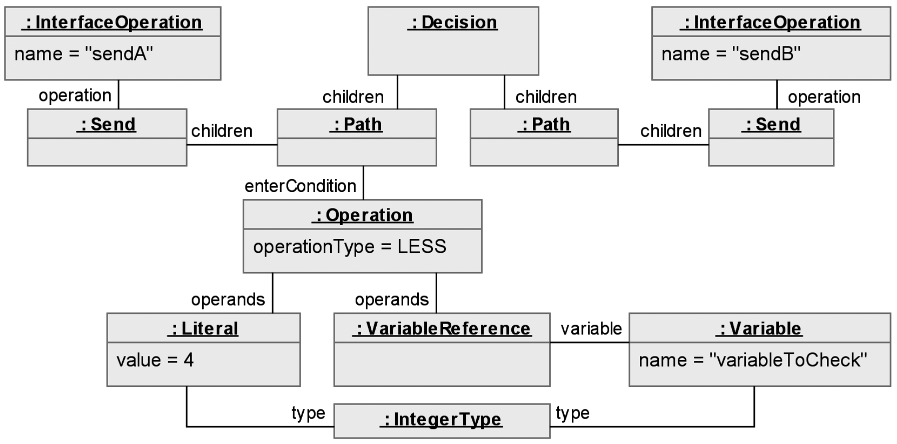


Figure 6.21: SMM2Java: Decision (SMM)

statement. Note that a special case occurs if no exit condition is given: In this case, the loop condition is set to `true` *and the loop iterates until the thread is interrupted.*

An example for a loop transformation is shown in figure 6.22 and listing 6.21, respectively.

```
do {
  partner.sendA();
} while (! (((Integer)getVariable("variableToCheck")) == 4));
```

Listing 6.21: SMM2Java: Loop (Java)

Finally, an SMM *Parallel* composite element requires starting the runnables corresponding to the individual paths and waiting for all of them to finish. This has been already been discussed in transformation step 17 and shown in listing 6.19.

Transformation 20 (Parallel) *The paths of a parallel statement are available as execution scopes and can thus be started and stopped as discussed in transformation step 17.*

The discussion of the transformation of structuring activities concludes the transformation from the Service Meta-Model (SMM) to the Java programming language.

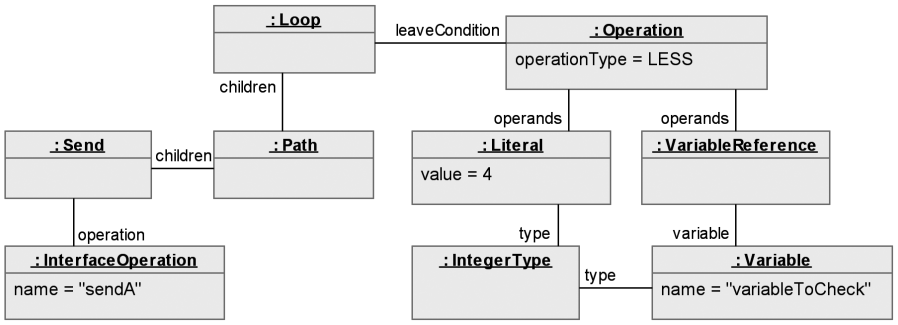


Figure 6.22: SMM2Java: Loop (SMM)

6.4.3 Semantics of SMM and Java

As in the previous transformation, the aim of the transformation from the SMM to Java has been enabling developers to ultimately convert from UML4SOA to code, while keeping to the semantics defined for UML4SOA discussed in chapter 5.

We believe that the above transformation creates Java implementations which are faithful to the MIO-based semantics. We further discuss this issue with a simulation and tracing approach in chapter 7.

6.4.4 Case Study Example

In the following, an overview of the generated Java code for the eUniversity case study already presented in the previous chapters is given. The static part consists of one class implementing the behaviour (as there is only one activity attached to the ThesisManager participant) with two provided interfaces (Student and Tutor), five required interfaces (Blackboard, Examination Office, Graduation Service, and again Student and Tutor), four message types (Thesis, Student, Grade, and Document) and one exception type (ThesisFailedException).

Furthermore, the generated code contains several helper classes mentioned in the transformation (in particular, `ServiceRunnable` and `BarrierRunnable`). The static structure of the converted eUniversity case study is shown as an UML class diagram in figure 6.23.

The behavioural part of the case study, consisting of the class `ThesisManager`, contains eight execution scopes (ThesisManager, Main, Registration, InProgress, StatusEventHandler, ExceptionHandler, RegistrationCompensationHandler, and finally the scope for the finished interrupting receive) and seven barriers (createThesis, acceptTopic, updateStatus, getStatus, getStatus_reply, finished, and getAssessment). The generated code consists of 322 lines and is thus not shown here. As an example, the code for the main `ThesisManager`

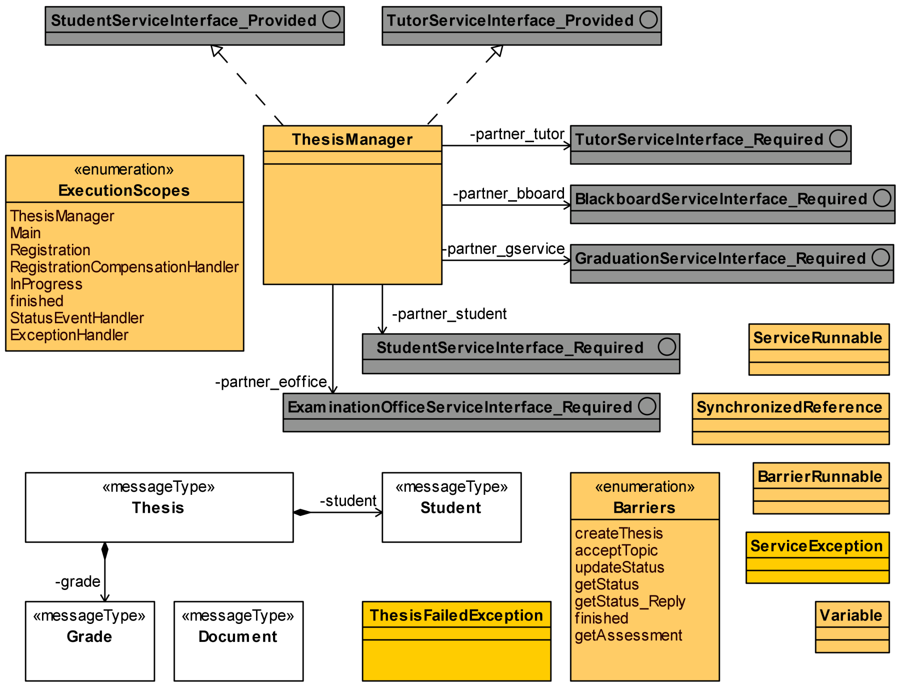


Figure 6.23: eUniversity Case Study: Static Part in Java

service runnable `run()` method is shown in listing 6.22.

The full implementation may be downloaded from the links provided in section 8.3.

6.5 Tool Support

In the previous sections, three model transformations starting from UML4SOA have been introduced. First, the UML2SMM transformation, which transforms from one platform-independent services model (UML4SOA) to another (the SMM); second, the SMM2WS and SMM2Java transformations, which transform from a platform-independent model (SMM) to platform-specific models; in the first case, to models of the various standards of the Web Service platform; in the second case, to a model of the Java programming language.

Describing the transformation in abstract terms is a good way of understanding the differences between the meta-models and therefore, the principles behind the transformation. However, such a description is not yet executable. This section describes the actual implementation of the three transformations listed above, and goes one step further by generating executable code from the generated target model instances.

```
public void run() {
  try {
    barriers.get(Barriers.createThesis).await();
    addVariable("thesis", new Thesis());
    ((Thesis) getVariable("thesis")).setThesisId((String) getVariable("thesisId"));
    ((Thesis) getVariable("thesis")).setTitle((String) getVariable("title"));
    ((Thesis) getVariable("thesis")).setDescription((String) getVariable("description"));

    partner_bboard.postToBoard((Thesis) getVariable("thesis"));
    runnables.get(ServiceActivity.Main).start(this);
    runnables.get(ServiceActivity.Main).waitForEnd();
    if (runnables.get(ServiceActivity.Main).getException() != null) {
      runnables.get(ServiceActivity.ExceptionHandler).start(this);
      runnables.get(ServiceActivity.ExceptionHandler).waitForEnd();
    }
  } catch (Exception e) {}

  setCanEnd(true);
}
```

Listing 6.22: eUniversity ThesisManager ServiceRunnable

6.5.1 Transformation Architecture

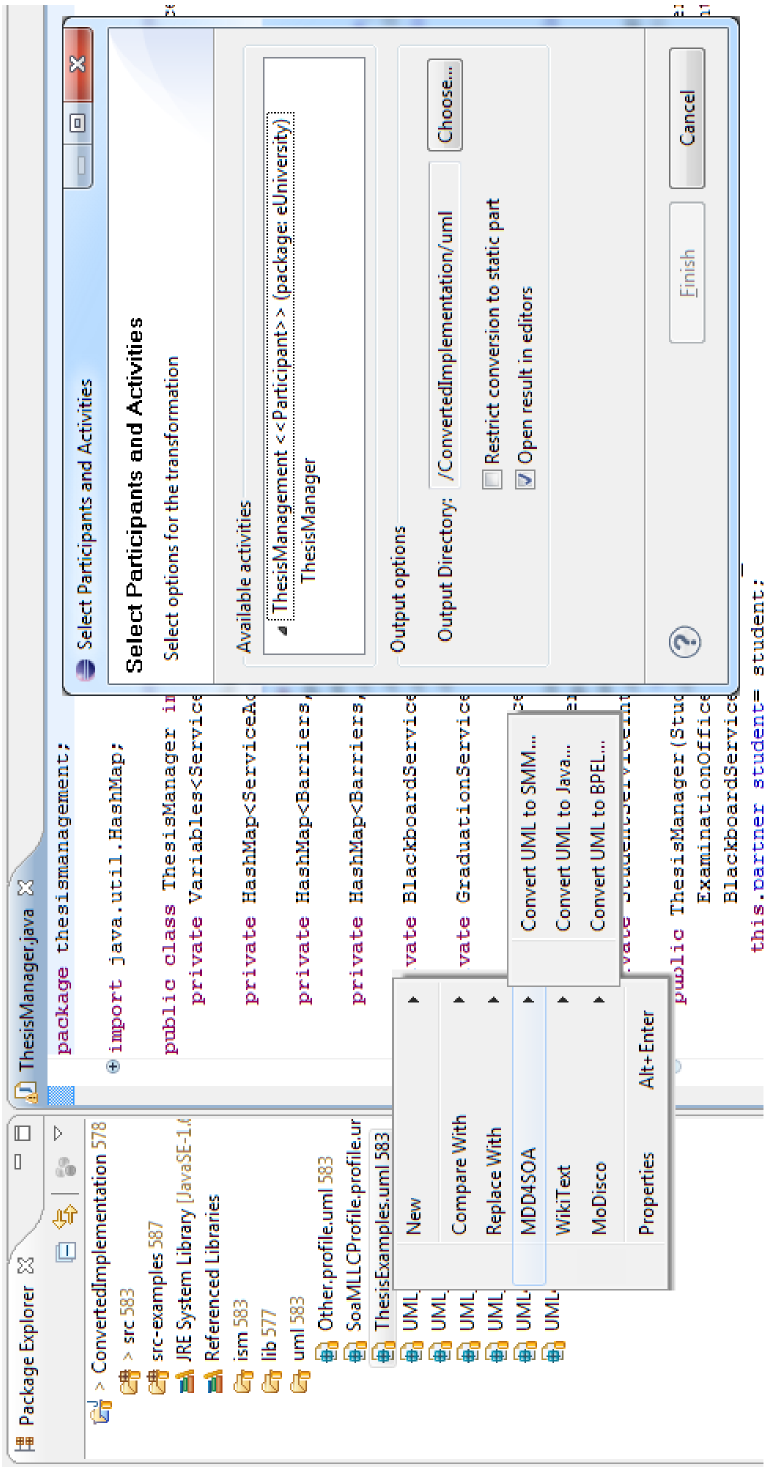
The model transformations described above are based on EMF models and their corresponding Ecore meta-models. Ecore is the EMF equivalent to EMOF, i.e. the root meta-model in which EMF itself and all EMF-derived meta-models are described. There are seven meta-models in use in the transformations:

- The source meta-model of the UML2SMM transformation is the *UML2 meta-model*. The actual meta-model implementation used in the transformation is the EMF-based implementation of the Unified Modeling Language (UML) 2 OMG meta-model for the Eclipse platform. The model is provided by the Eclipse Model Development Tools (MDT) [Ecl10c] project. This meta-model is a complete implementation of the UML2 standard and thus includes all static and behavioural elements of the UML.

- The target meta-model of the UML2SMM transformation, which is also the source model of the SMM2Java and SMM2WS transformation, is the SMM meta-model which underlies modelling, analysis, and code generation in this thesis and has been described in section 4.

- The target meta-model of the SMM2Java transformation is the MoDisco EMF model [Ecl10e]. The MoDisco Java meta-model it a reflection of the Java language, as defined in version 3 of the Java Language Specification [GJSB05].

- Finally, there are several target meta-models in the SMM2WS transfor-



```
package thesismanagement;

import java.util.HashMap;

public class ThesisManager im
    private Variables<Service
    private HashMap<ServiceAc
    private HashMap<Barriers,
    private HashMap<Barriers,
    vate BlackboardService
    vate GraduationService
    public ThesisManager(Stud
        ExaminationOffice
        BlackboardService
        this.partner student= student;
```

Select Participants and Activities

Select options for the transformation

Available activities

- ThesisManagement <<Participant>> (package: eUniversity)
  - ThesisManager

Output options

Output Directory: /ConvertedImplementation/uml

☐ Restrict conversion to static part

☑ Open result in editors

Figure 6.24: Transformation Tool in Eclipse

mation. Being part of the Web Service family of standards, BPEL closely integrates with the Web Service Definition Language (WSDL), the XML Schema Language (XSD), extensions to WSDL (Partner Link Specifications and Correlation Property Specifications) as well as the SOAP and WS-Addressing standards. The corresponding meta-models are provided by the Eclipse Web Tools Platform [Ecl10j] (WSDL, SOAP), the Eclipse Modelling Tools (XSD) [Ecl10c] and the Eclipse BPEL project (BPEL) [Ecl10g].

Recalling from chapter 2, a model transformation translates between two models. Additional steps are required for acquiring the source model from its native representation and emitting the code corresponding to the target model, as shown in figure 2.12 on page 39.

Each of the meta-models provided by the various projects listed above provide deserialisation- and serialisation support as required for the model transformations. The former is relevant for the transformation sources: The UML2SMM and SMM2Code transformations must be able to read the source model instances from a stored representation. In both cases, this is a specific XMI format defined by EMF.

The latter is relevant for the transformation targets: Here, the aim is creating the actual (source) code of a model instance. In the case of Web Services, for example, the serialisation is provided in the (XML-based) BPEL language and its required elements (WSDL, XSD, WS-Addressing, etc.). In the case of Java, the serialisation creates actual Java code; the MoDisco model employs a model-to-text transformation written in the Model to Text (MTL) [OMG08b] language defined by the OMG and executed by Acceleo [Ecl10h] to emit Java code.

In most cases, the user of these transformations is not interested in the technical details behind the scenes. Thus, the transformation tools have been integrated into the Eclipse platform where they can be invoked on the relevant input UML artefacts, directly generating Web Service or Java code. This is discussed in the next section.

6.5.2 Tool Integration

The transformation steps between two instances of the corresponding meta-models shown in the middle of figure 2.12 are implemented as Eclipse plug-ins, and thus contribute their functionality to the Eclipse workspace.

Thus, starting from a .uml file inside the Eclipse workbench, developers can select the targets BPEL and Java, and additionally (for testing purposes), the SMM. Upon selecting a target, the developer is presented with a choice of which participants and behaviours to convert. Selecting OK then transforms the artefacts; the result is shown in the workspace as well and can be opened for inspection by the user.

An example of the UI is shown in figure 6.24. On the left-hand side, the resource explorer is shown with several .uml files; on one, the transformation has been invoked. In the centre, the selection dialog is shown which enables users to select individual participants or behaviours from the model. In the background, a converted Java file is shown.

The transformation plug-ins are available online. The corresponding links can be found in section 8.3.

6.6 Related Work

As the UML4SOA profile is a product of this thesis, there is expectedly no related work which describes transformations from UML4SOA to executable code. However, there are two interesting related areas which we would like to present here.

First, Chapter 3 has already mentioned some alternative UML profiles for behavioural SOA modelling, some of which have model transformations attached. These are discussed again, with a focus on transformation, in section 6.6.1.

Second, there are a number of related transformations which can be used in combination with the MDD4SOA transformers. Two concern the related SENSORIA profiles discussed in section 3.5.2; two have been written as follow-ups to the transformations of MDD4SOA. These are listed in section 6.6.2.

6.6.1 From SOA UML Profiles to BPEL

In this section, we discuss UML-based profiles with support for behavioural modelling of service behaviour and transformation support. This is a subset of the profiles discussed in section 3.5.

The first explicit attempt of converting a UML-based SOA structure with behaviour to BPEL known to us is the work of Skogan et al. [SGS04]. The transformation from workflow to BPEL is an XSLT transformation. It explicitly addresses communication between services; however, higher-level concepts such as compensation, exceptions, or events are not mentioned.

The UML-S profile [DNsmGW08] was also extended with transformation support. In [DGW08], the authors have provided an overview of rules for transforming UML-S behaviour to BPEL. The focus lies again on workflow patterns; communication and again more complex constructs such as compensation and events are not addressed. Also, tool support is not mentioned for this translation.

In their work on a UML2 profile for service modelling [EK07], Ermagan and Krüger mention transformations as a possible way of realising the UML behaviour, but no further elaboration is made.

Finally, the BPEL-specific profiles mentioned in section 3.5 also include transformation support. The first profile [AGGI03] can be mapped to WSDL and BPEL by using the (commercial) products Rational Rose and Rational XDE. The document lists compensation as future work. However, a follow-up diploma thesis lifts the profile up to BPEL 2.0 and provides a complete transformation using non-commercial technology.

The second profile [Man03] also includes transformation support; however, it is only shortly described and tool support is no longer available.

The approach by Li et al. [LZP09] discusses automated generation of BPEL from UML sequence diagrams annotated with BPEL-like stereotypes. The process starts with a WSDL file which is used as the functional description of each service. As the sequence diagram may contain more than two life lines, it is possible to get a more global perspective than using activity diagrams. The paper does not mention faults, events, or compensation.

In general, however, the basic problem of these profiles discussed in chapter 3, i.e. their closeness to BPEL, remains.

It should be noted that we do not know of any other attempts to transforming behavioural SOA models directly to plain Java. Still, we believe that this *compiler-approach* is an interesting topic and should be investigated further.

6.6.2 Related Transformations

Two of the additional Sensoria profiles discussed in section 3.5 enjoy transformation support. The first is the non-functional properties extension; the second is the architectural modes extension.

With respect to non-functional properties, NFP-enriched UML models can be used to automatically generate Quality of Service (QoS) artefacts for the Web Services family of standards, which is shown in [GGK$^+$10]. The transformations handle reliable message communication and security in service-oriented systems, and include both the generation of a WS-Policy-compliant descriptor (with links to WS-Security and WS-Reliable Messaging), as well as platform-dependent configuration files for middleware platforms such as Apache Sandesha.

The second profile deals with service modes, which describe different architectural configurations of a SOA-based software system. A transformation is available [FUMK08] which generates a series of dynamic service brokering requirements and capability specifications for the service broker Dino [MDEK95]. The functional description of a service taking part in a mode is described using OWL-S; the constraints for service brokering which includes QoS attributes is described in an XML document to be read by Dino.

The Sensoria project has included several case studies, among them the finance and the automotive case study, which have been introduced by two separate companies (s&n and Cirquent, respectively). Each company deals with different target platforms; the first with the ActiveBPEL engine [Act10], the second with the JBoss jBPM application server [JBo10]. Both, however, have faced the problem of adding user interactions (i.e. a human component) to the BPEL processes generated by MDD4SOA. Thus, two transformations have been written which take the generated BPEL output from the MDD4SOA transformers as input and provide a human-enabled, container-specific version as output.

Each transformation solves the above problem by adding a dedicated Web Service as an intermediary between the user and the business process; in each case, a web-based UI is created for user interaction. Several activities are added to the original business process which enable communication between the intermediary and the business process itself.

The first of these transformations is the ActiveBPEL transformation created by Cirquent for the automotive case study [XK09] of SENSORIA. It includes the `ViewManager`, a service-side program to coordinate the BPEL process and the user interfaces. The transformations adds a series of invoke and reply statements to the main BPEL process which allows it to interoperate with the `ViewManager`. Note that the `ViewManager` is tied to the ActiveBPEL engine as it has access to and depends on the receive queue of the BPEL process for deciding what to show next in the UI. The transformation also includes the generation of deployment artefacts for the ActiveBPEL engine.

The second transformation is the task manager service transformation created by s&n for the finance case study [ES09] of SENSORIA. A task system [Lin06] is used to bridge the gap between users and BPEL process, which is based on *tasks* which are made available to and need to carried out by humans via a Web site. A dedicated Web Service — the `TaskService` — is used implement this bridge. The task service is specifically generated for each BPEL process and, as the name suggests, is bound to a Web site on which the different tasks are displayed and thus available for completion by various kinds of users (such as a customer, a clerk, or a supervisor). Again, the original BPEL process is extended with invoke and reply statements for keeping the task manager up to date. Like before, the transformation includes the generation of deployment artefacts for the jBoss application server.

As noted above, the last two transformation discussed go beyond what has been provided by the MDD4SOA transformers: We have left the issue of human interaction open, as it does not directly affect the service specification. Thus, these two transformations give a good insight on how one might proceed in an automated way from the artefacts generated by MDD4SOA.

6.7 Summary

This chapter has introduced the code generation part of model-driven development with UML4SOA, which enables UML4SOA modellers to automatically convert their models to either the Web Service technology stack (which includes BPEL, WSDL, and XSD artefacts) or the Java programming language.

The first section (section 6.1) of this chapter has discussed the use of *model transformations* for converting UML4SOA models to source code. Transforming models from UML4SOA to standard executable target languages provides some challenges: the static, behavioural, and data handling parts need to be considered. As UML4SOA has the backing of the Service Meta-Model (SMM) introduced in chapter 4, the transformations have been split into two parts; first parsing the UML source model into an instance of the SMM and only then moving on to the target languages.

In the following, three model transformations have been discussed. First, section 6.2 has introduced the UML2SMM transformation, whose main focus is recognising patterns in the UML4SOA activities and partitioning the service activities to create a well-nested, ordered structure to be represented in the SMM. Furthermore, this transformation deals with parsing data statements, converting them to an instance of the AST-like SMM data part.

Section 6.3 has shown a transformation from the SMM to the Web Service family of standards, which includes BPEL, WSDL, XSD, and accompanying standards. While the main structure of the SMM and BPEL is quite similar, a lot of detail in handling the individual artefacts is required in this transformation to ensure executability by a BPEL engine. Furthermore, some of the concepts of the SMM are non-existant in BPEL and must be added manually.

The third transformation SMM2Java has been discussed in section 6.4. As Java is an object-oriented language without built-in SOA support, an adequate transformation concept has been introduced first. The transformation uses this concept, which is based on threads, thread nesting, and barriers, to create an appropriate representation of SMM participant behaviour in Java, staying true to the spirit of the SMM but using as many natural Java concepts as possible.

Afterwards, section 6.5 has given a short overview of the tool support for the transformations described and their integration into the Eclipse workbench. Each transformation is fully implemented and executable.

Finally, section 6.6 has discussed related work.

Chapter 7

Simulation and Tracing

In this chapter, we investigate the runtime behaviour of UML4SOA participants based on the generated Java code introduced in chapter 6, comparing this behaviour to the MIO-based semantics of UML4SOA shown in chapter 5. We are thus addressing the fourth contribution area of the MDD4SOA approach shown in figure 1.1 (introduction chapter, page 3): Simulation based on the generated code on the right-hand side as well as trace annotation from the generated code to the formal model.

The chosen approach is based on simulation, tracing, and trace comparison. Firstly, we introduce an automated testing environment which allows direct execution of the generated code and thus also the UML4SOA and SMM models. An execution generates a trace consisting of send and receive events as they occur during runtime. Through the introduction of latency, the execution time is stretched at various points, thus simulating different load conditions. A multitude of such traces is generated to ensure a high coverage of the generated code.

Secondly, the generated traces can be used to gain insight into the runtime behaviour of the code: The series of events present in a trace can be annotated on the formal model by symbolically executing the corresponding modal input/output automata. This yields statistical information on how the paths present in the automata are actually used during runtime. Additionally, this information can also be used for validating the implementation.

An overview of our approach is given in section 7.1. Simulation and the generation of an automated testing environment is discussed in more detail in section 7.2, followed by tracing and trace annotation on the formal semantics in section 7.3. We apply our approach to the eUniversity case study in section 7.4. Tool support for both simulation and annotation is discussed in section 7.5, and a summary is given in section 7.6.

Published results: The results presented in this chapter are original and have not been published before.

7.1 Introduction

This chapter introduces a method for automatically executing the generated code of a participant behaviour, and using trace information from such executions for understanding and comparing the runtime behaviour against the formal semantics.

First, an automated testing environment is used for creating a trace of events which each denote a sending or receiving operation in the code. Second, this trace can be compared with the formal semantics of the corresponding participant behaviour, yielding information about how the paths of the automata are actually used during runtime. This also allows validation of the code against the formal semantics: If the trace can be consumed by the automata, the execution of the generated code conforms to the specification; else, it does not.

Figure 7.1 shows the simulation and tracing approach.

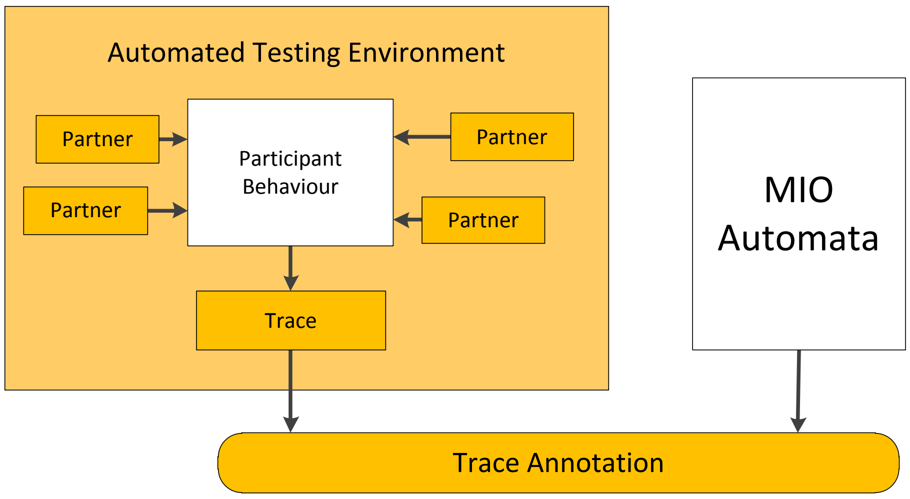


Figure 7.1: Simulation and Tracing Approach

The approach consists of three main components:

- Firstly, the generated code needs to be executed. We thus introduce an *automated testing environment* which is able to call and receive operations of the participant behaviour by simulating any external partners attached to the behaviour.
- Secondly, the generated code will need to keep a record of its activities, i.e. write a *trace* of events occurring in the code.
- Thirdly, we can compare the generated traces to the modal/input output automata representing the semantics of the behaviour with a *trace comparison* component.

As each individual trace only represents one path through the executed code, it is furthermore important to generate many different traces to achieve a high coverage of the generated code. A common way of achieving different traces in testing of concurrent systems is artificially slowing down the implementation, creating arbitrary latency at different places to elicit different behaviour (or at least, different code paths taken). We use this technique in our approach as well.

Our simulation and tracing approach has several benefits for developers of SOA systems using the MDD4SOA approach:

- Firstly, the automated testing environment — which is automatically generated from the model — can be directly started and thus employed to test the model, thus following the basic idea of the model-driven approach. The testing environment can not only be used for testing the generated code, but also changes to the code as well as completely custom implementations.

- Secondly, results from such an execution — in the form of observed behaviour or in form of a trace — gives feedback to the developers which can be used to improve the model. In particular, traces can be compared against the formal model, which yields information about which parts are used in the implementation.

- Finally, the trace annotation approach can also be used to validate any executed code, as it yields an error if a trace can not be consumed by the modal input/output automata.

We have chosen the Java implementations generated by SMM2Java for our simulation approach. Java is a good choice for this approach for the following reasons:

- With Java being an object-oriented language, all SOA aspects have been implemented on top of the language. The implementation thus directly follows the semantics of the SMM and UML4SOA.

- The Java implementation is very flexible with regard to adding statements required by the approach, such as logging statements for denoting events as they occur.

- The tools used to implement the approach — the SMM2Java transformer and the Mio Workbench — are based on Java and Eclipse, and the resulting code may be run within Eclipse as well. Thus, we get integrated tool support for all steps.

In the following, we first discuss the simulation aspect of the approach. Afterwards, we introduce trace generation and comparison, i.e. a method for checking the generated traces against the formal semantics.

7.2 Simulation

The simulation aspect of our approach is made possible by the generation of an automated testing environment which is able to execute the participant behaviour. As this behaviour expects calls from its partners on provided interfaces and sends out calls to required interfaces, the testing code must be able to invoke the operations offered by the participant as well as be available for invocations from the participant. Thus, the testing environment must implement *all* partners of the participant.

As the aim is the complete execution of the participant, the testing environment must also follow the communication rules which are specified as protocols of the partners of the participant. Thus, contrary to the transformation of *participant behaviour* from the SMM to Java, we now face the challenge of transforming *service protocols* to Java as well. As each partner is fundamentally independent from all others, they should be executed in parallel, each following its own generated representation of the `ServiceProtocol` state machine given in the SMM.

In each state, the partner may be faced with one of three situations: Either the partner needs to invoke an operation, receive an operation, or do both:

- *Send State.* The state may have only outgoing send actions for the partner. In this case, the implementation can freely (and non-deteministically) decide which transition to take.

- *Receive State.* The state may have only incoming receive actions for the partner. In this case, the implementation waits for one of the messages to arrive. To avoid waiting forever, a timer is necessary which aborts a receive after a certain time, leading to an error.

- *Mixed States.* In case both send and receive transitions are possible, the partner must be able to receive a call. The partner may also decide to arbitrarily send out a message, in which case the receiving process will be aborted.

Note that as all protocols are seen from the point of view of the participant, the transition roles are inverted: A `ReceivingTransition` leads to a send operation in the partner, while `SendingTransition` leads to a receive.

Besides the actual implementation of the protocol state machines, the testing environment must also include statements for starting and stopping partners, and for connecting the participant code with the environment.

As mentioned above, single traces generated from an execution only represent individual paths through the executed code. To achieve a more thorough picture of the possible traces, a large number of traces must be generated; each run should furthermore ideally elicit different behaviour from the code. There are two ways in which we can change the path through the generated participant behaviour:

- Firstly, as indicated above, the implementation may be *slowed down* by introducing latency at various points in the implementation. Latency should be chosen non-deterministically in each execution; thus, each run exhibits different timings, increasing the chance for execution of a different path. We add latency by means of *wait* statements.
- Secondly, participant implementations may also exhibit different behaviour based on different input data. This touches the area of test case generation and model-based testing, which lies outside our focus. We thus have opted to simulate only one set of data, which consists of the default data values of Java (for example, `0` for an `integer`, `true` for a `boolean`, and a new instance of more complex types). If other paths are necessary, the testing environment needs to be adapted by hand.

The testing environment may be generated from the SMM with the following transformation steps. The first creates the central simulator class and initialises the partners.

Transformation 21 (Testing Environment) *For each behaviour of a `Participant`, a simulator class is created. The class carries the same name as the behaviour with the `Simulator` suffix. It contains a main method which instantiates the class and invokes the `start` method. The `start` method in turn initialises the behaviour class in a field, and starts a thread for each partner (with the partner runnables, see below).*

The partners are bootstrapped by executing the following step for each partner of the participant to which the behaviour is attached:

Transformation 22 (Simulated Partners) *For each partner of the current participant, a) an enum is created, containing the states in the corresponding partner protocol, b) a `currentState` field is created for the current state value, and c) a queue is created for incoming messages for this partner.*

We need to ensure that we are able to receive all messages sent to the testing environment as well as sending all possible messages out of the testing environment; thus, a method for each send and receive is created. The former reside in the simulator class, the latter in anonymous class implementations of the corresponding partner *required interface.*

Transformation 23 (Send Methods) *For each implemented method of a provided service and each used method of a required service, a method with the operation name and a `send` prefix is created in the simulator class. The send method invokes the corresponding method on the actual participant class field.*

Transformation 24 (Receive Methods) *For each partner which has a required interface, an anonymous subclass of this interface is created and stored in a field. Inside the subclass, all methods of the interface are implemented. Each method stores the operation call name in the queue of the corresponding partner.*

We can now turn to the actual implementation of the partners. As said above, each partner is implemented as a thread; thus, a runnable is created for each partner, which contains a loop and a switch over all states.

Transformation 25 (Partner Runnables) *For each partner, a runnable is created. The `run` method uses a `while` loop with a switch statement to select the current state. Each case decides what is to be done based on the outgoing links of the current state; the loop is left if a state has no outgoing links.*

In each `case`, we add a `wait` statement which introduces an arbitrarily selected latency[1]*.*

We have discussed the issue of three different state types above. These are converted as follows.

Transformation 26 (States) *For each state in a partner state machine:*

- *If there are only outgoing sends, one is picked non-deterministically. The corresponding operation is called and the next state is selected.*
- *If there are only incoming receives, we wait for one of them to arrive, thus advancing to the next state. If a timeout occurs while waiting, or a wrong message is received, the partner thread is closed.*
- *In mixed states, we again wait for incoming receives. During the waiting time, the implementation may decide non-deterministically to send a call instead. In both cases, the impementation moves to the next state as indicated by the corresponding transition.*
- *In case a state does not have any outgoing links, the partner thread is closed.*

This concludes the generation of the automated testing environment. The simulator class can now be directly started; it will interact with the generated participant behavioural implementation. Due to the generated timeouts, the testing environment will terminate in any case, which includes both normal situations (for example, optional messages may not arrive) and error situations (wrong message order, or exception).

As an example, listing 7.1 shows the code for a part of an eUniversity partner (`student`). In `State_1_working`, the partner sends out a message — either `updateStatus` or `finished`. The state moves to `State_1_working` in the first and `State_2_finishing` in the second case.

In `State_2_finishing`, a receive is shown: We need to wait for a message (`assessmentComplete`) to be received. In case it does not arrive in time (timout), an error is reported. If the wrong message arrives, an error is reported, too (protocol breach).

[1]In our implementation, latency lies between zero and one seconds.

```
enum States_student { State_0_started, State_1_working, State_2_finishing, State_3_done }

Runnable partner_student= new Runnable() {
  public void run() {
    partner_state_student = States_student.State_0_started;
    while (true) {
      switch (partner_state_student) {

        ...
        case State_1_working:
          SimHelper.waitRandom();
          int rnd = new Random().nextInt(2);
          switch (rnd) {
            case 0:
              send_updateStatus();
              partner_state_student = States_student.State_1_working;
              break;
            case 1:
              send_finished();
              partner_state_student = States_student.State_2_finishing;
              break;
          }
          break;

        case State_2_finishing:
          SimHelper.waitRandom();
          int timer_finishing = 0;
          while (queue_student.peek() == null
            && timer_finishing <= TIMEOUT)
              timer_finishing += SimHelper.waitRandom();

          if (timer_finishing > TIMEOUT)
            // return with error (timout)

          String msgReceived_finishing = queue_student.poll();
          if ("assessmentComplete".equals(msgReceived_finishing)) {
            partner_state_student = States_student.State_3_done;
            break;
          }

          // return with error (wrong message)

        ...
        case State_3_done:
          return;
      }
    }
  }
}
```

Listing 7.1: Simulation: Testing Environment Code

7.3 Tracing

To be able to gain information from the execution of the behaviour, we need to log the series of events handled within the service behaviour. Second, we can use this information for comparison with the formal model. We discuss each step in turn.

7.3.1 Generating Traces

Trace generation is made possible by the addition of logging statements to all send, receive, reply, and send&receive actions in the generated code. During this process, additional opportunity for different execution paths can be created by adding latency-generating statements to the implementation as well.

Chapter 6 has already introduced transformation steps for generating the individual communication actions mentioned above. We can extend them here to emit trace logs and add latency. This touches transformation steps 13, 14, and 15. The first and the second two are extended as follows, respectively:

Transformation 27 (Send and Send&Receive) *Before the Java method call on a partner, a trace statement is added which stores the current time, partner, operation (*send*) and message sent in an appropriate place for later retrieval. In case of a* send&receive*, another trace statement is added after the method call which denotes the return of the method. The second trace statement thus stores a* receive *event. Furthermore, a* wait *statement is added to simulate latency.*

Transformation 28 (Receive and Reply) *For receive and reply operations, barriers are used in the implementation. In each barrier, a trace statement is added which stores the current time, partner, operation (*receive*) and message received in an appropriate place for later retrieval. Finally, a* wait *call is added again.*

Having trace statements in place generates a series of trace messages for each run of the environment. Each trace message either denotes a message sent or received. An example of a trace (taken from the eUniversity case study) is shown in listing 7.2: Each trace event is annotated with time, event, partner, and the operation sent or received.

7.3.2 Trace Comparison

Each run of the testing environment yields a trace of send and receive events with attached messages. The message order can now be compared with the modal input/output automata (MIO) which defines the semantics of the corresponding participant behaviour. This is done by symbolically executing the automata with a *trace annotation* algorithm. The algorithm begins in the start state of the automaton with the first event in the trace, and proceeds as follows:

```
17:02:33:698:RECEIVED:tutor:createThesis
17:02:33:700:SEND:bboard:postToBoard
17:02:34:329:RECEIVED:student:acceptTopic
17:02:34:329:SEND:bboard:removeFromBoard
17:02:34:331:SEND:eoffice:thesisStarted
17:02:34:331:SEND:gservice:registerForGraduation
17:02:34:332:SEND:tutor:thesisInProgress
17:02:34:378:RECEIVED:student:updateStatus
17:02:34:758:RECEIVED:student:finished
17:02:34:758:SEND:tutor:getAssessment
17:02:34:758:RECEIVED:tutor:return_getAssessment
17:02:34:759:SEND:student:assessmentComplete
17:02:34:759:SEND:eoffice:thesisPassed
```

Listing 7.2: Example Trace

- In each state, the outgoing transitions which can handle the current event are identified. There may be more than one — for example, an internal transition and a receiving transition with the same operation name as an receive event might be possible. In this case, one transition is chosen non-deterministically[2]. If the chosen transition option is later found to lead to an error state, the next one is chosen.
- The transition is taken, i.e. the current state moves to the target state of the transition. If the transition is external, the next element from the trace is selected; otherwise, the current trace element stays the same. The algorithm continues recursively with the new state and trace element.
- Aborting conditions:
  - If the end of the trace is reached, the execution was successful. A list of executed transitions is returned.
  - If a state does not have outgoing transitions, or none matches the current element, the algorithm returns to the previous level and chooses the next possible path.
  - If all options are exhausted without reaching the end of the trace, an error is reported along with an error situation, i.e. a path taken which leads to a situation where an event cannot be consumed.

Symbolically executing the automata can either be successful (i.e., all messages can be consumed by the automata) or not (i.e., one message is not possible in a certain state). While the former indicates that, for this trace, the implementation in Java conformed to the semantics, the second indicates that a certain trace was possible in the implementation although not allowed by the semantics.

[2]Enumerating all possible ways through an automaton yields too many paths, even with small examples.

Note that a successful execution does not prove that the implementation is free of errors; however, it does provide some empirical evidence.

The output of the trace annotation algorithm is a path through the automaton, which consists of alternating states and transitions. Later, this path can be shown as a graphical annotation on the automaton — in case the trace was not fully consumed, including the information about an event which was not possible in a certain state.

We can now refer back to the trace shown in listing 7.2, which is an example of a successful trace. A graphical representation of (parts of) this trace is shown in figure 7.2: The corresponding path has been selected and is drawn in red; all others are merely possible options which have been ignored.

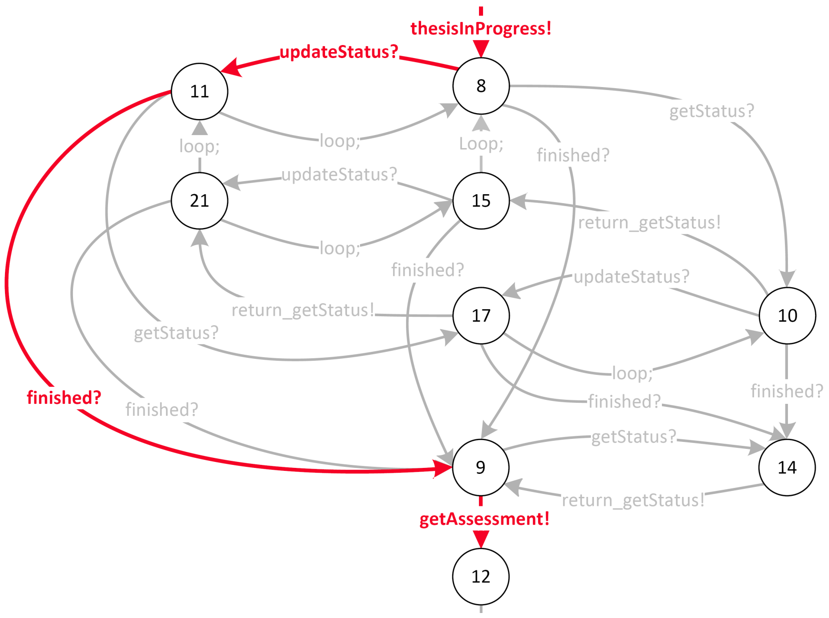


Figure 7.2: Trace Annotation

In case the trace includes a problematic sequence, the annotation is done as far as possible (i.e., until the first non-solvable situation). For example, consider switching the `finished` / `getAssessment` calls in the listing above. In this case, the annotation would end at state 11, indicating that `getAssessment` was not possible in this state.

7.4 Case Study

We have applied the simulation and tracing approach to the eUniversity case study. Generating the code for the case study yields 480 additional lines of code for the simulator class as well as 16 additional log- and wait statements, respectively, in the actual participant code.

As mentioned above, the generated code contains only primitive handling of data, i.e. newly created instances of each message type. However, the eUniversity participant behaviour has two possible paths, depending on which grade the tutor selects. We have thus adapted the testing environment code into two versions: The first version gives a grade of 1 (which is smaller than 4), the second of 5 (which is larger than 4 and means the student has failed). We have then generated 10^4 traces of each version, and have annotated these traces onto the MIO representation of the participant shown in figure 5.8 on page 159.

The annotation has shown *no errors*. A statistical analysis of the executed transitions shows that all transitions except for the final one (which is a technical transition showing that the compensation handler was executed) have been executed at least once. Some of the transitions — in particular, the initial ones and the `getAssessment` call to the tutor — have been executed in each trace, and thus $2 \cdot 10^4$ times. Within the `InProgress` scope, numbers vary as the `updateStatus` call as well as the `getStatus` and `return_getStatus` transitions are optional, and there are multiple transitions for each event due to interleaving and the event handler loops. We have annotated the statistical result graphically in figure 7.3: Each transition is associated with a color from red to blue. Full red indicates a *hot* transition (i.e., executed every time) while full blue indicates a *cold* transition (i.e., never executed); these are combined in 23 gradations.

7.5 Tool Support

Simulating Java code and comparing the result on MIOs is made possible by extensions to the SMM2Java transformer introduced in chapter 6 as well as the Mio Workbench introduced in chapter 5.

Firstly, the SMM2Java transformer includes both the transformations steps discussed in sections 6.4 and section 7.2. The UI for the transformer allows to enable the generation of the automated testing environment, which means creating a simulator class as well as the necessary logging- and latency statements in the actual implementation.

Secondly, the Mio Workbench includes a graphical view of a MIO which can be extended with trace annotations. The Mio Workbench is able to read both single trace files as well as entire directories of traces. In the first case, the trace is annotated on the view; if the trace contains invalid actions this is shown to the user in a dialog box. In the second case, all traces are checked versus the automata. In case all traces are valid, a statistics file is generated; otherwise, the first problematic trace is annotated onto the MIO representation.

A screenshot of the Mio Workbench with an annotated trace is shown in figure 7.4.

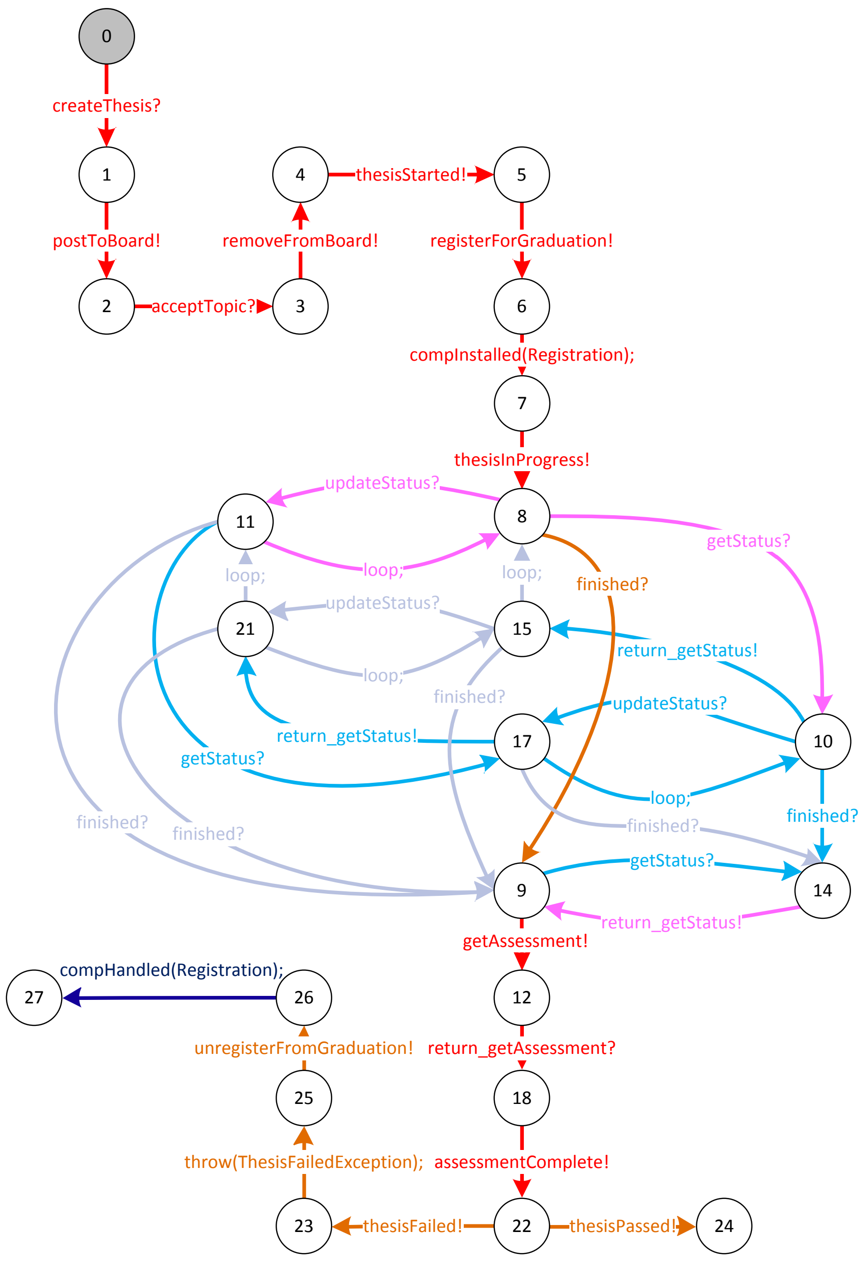
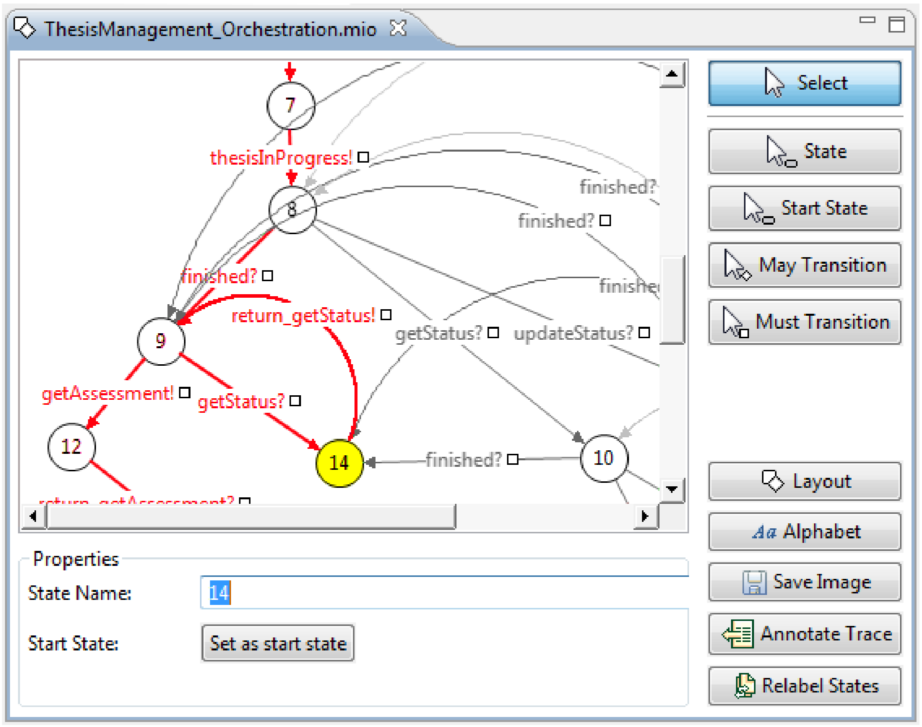


Figure 7.3: eUniversity Full Trace Annotation



State Name: 14

Start State:

Figure 7.4: Trace Annotation in the Mio Workbench

7.6 Summary

This section has introduced an approach to simulation and tracing of generated Java implementations of SMM models, and thus, in turn, of UML4SOA models. Furthermore, we have shown how to compare traces to the automata of the MIO-based formal semantics.

We have begun by discussing the generation of an automated testing environment in addition to the actual participant behaviour which is based on the protocols associated with each partner of the participant and executes different paths in the implementation by employing non-deterministically chosen send operations as well as the introduction of latency (section 7.2). The testing environment can be used on the generated code, but also on manually created or changed code.

Secondly, we have discussed how to augment the participant behaviour itself with logging statements such that an execution of the behaviour produces an output trace, and how to add additional latency to the implementation.

In section 7.3, we have discussed how to symbolically execute an automaton given a trace, which yields a path through the automaton. This path yields information about actually executed transitions of the automaton, and can also be used for validating the behaviour of the code.

Our approach has several benefits for developers. Firstly, the testing environment provides the ability to directly execute the generated code and thus an UML4SOA model. The testing environment can also be used for checking manual changes, or completely custom implementations. Secondly, trace generation allows developers to understand how the implementation proceeds through the different paths of the corresponding automaton at runtime. Finally, the traces can also be used to validate the code against the semantics.

We have applied the approach to the eUniversity case study in section 7.4 to show how the generated code compares to the MIO semantics, which has shown that all parts of the automata are in fact used during runtime, and no errors occur with regard to the semantics. Finally, we have shown how the MDD4SOA tools have been extended to deal with simulation and tracing (section 7.5).

Chapter 8

SOA Tooling for SOA Software

Developing service-oriented software involves dealing with multiple languages, platforms, artefacts, and tools. In the previous chapters, we have introduced tool support for the concerns of modelling, analysis, and code generation. However, when considering the complete software development process, additional tasks like policy specifications, artefact integration, deployment, and runtime support need to be considered. There are tools available from both academia and industry for addressing these concerns.

A tooling platform specifically targeted at the integration of tools for the service-oriented development process can provide support for developers to find, use, and combine all of these tools. In this chapter, we introduce the Service Development Environment (SDE), a service-oriented tooling platform for developing service-oriented software. The SDE does not only include the tools discussed in this thesis or developed within the Sensoria project, but also external tools from both academia and industry. It is furthermore based on the Eclipse platform which ensures interoperability with a wealth of other tools.

In section 8.1, an overview of tool integration for SOA development and the support by the SDE is given. Section 8.2 details the design of the integration platform. In sections 8.3 and 8.4, we give an overview of tools (specifically created for MDD4SOA; and additional ones) integrated into the SDE. Section 8.5 shows examples of how tools can be orchestrated to perform in collaboration. Finally, section 8.6 discusses related work, and section 8.7 concludes the chapter.

Published results: Results presented in this chapter are based on publications [MB07], [MRH08], [MR10], [FM08], [WHK$^+$08], [WHA$^+$08], and [AKCF$^+$08]. Furthermore, the SDE is a result developed as an answer to one of the main Sensoria research objectives (comprehensive tool support) and has been reported in several technical reports, brochures, and presented at fairs.

8.1 Integrating Tools for SOA Development

The success of the Service-Oriented Architecture (SOA) [Erl05] in both industry and research has resulted in a growing need for tool support for developers of services and service-based systems. Specific support for developing SOA systems is beneficial in all phases of the development process, ranging from modelling to runtime, from analysis to implementation.

The previous chapters have introduced tool support for three important areas in the development of service-oriented software systems:

- Chapter 3 has introduced a UML profile for modelling the behaviour of SOA systems, including support for two industry-standard modelling tools (MagicDraw and Rational Software Artefact).
- Building on these specifications, chapter 5 has introduced formal analysis support for UML4SOA models by means of a transformation tool to the domain of modal input/output automata, and the Mio Workbench for the verification of the transformed models. Additionally, tool support for re-annotation of UML models has been discussed.
- Finally, chapter 6 has introduced tools for transforming UML4SOA models to artefacts based on the Web Services standards platform as well as the object-oriented language Java.

From a developers standpoint, it is important to be able to use these tools in combination to ensure a smooth transition from one step to another, thus increasing productivity and control over the development process. As developers might also want to use other tools in the same process, tool integration should be open and standards-based in order to allow easy inclusion of other development tools.

These considerations have led to the development of a tooling platform, the Service Development Environment[1] (SDE) [MR10], which integrates the various tools required in the service development process, including modelling, analysis, code generation, and runtime functionality. The SDE gives developers an overview of available tools and their area of application, and allows developers to use tools in a homogeneous way, re-arranging tool functionality in a common way as required.

Through integrated tools, the SDE supports the following areas of the development of SOA systems:

- *Modelling:* Graphical editors for familiar modelling languages such as UML, which allow intuitive modelling at a high abstraction level, and also text- and tree-based editors for formal languages.
- *Model Transformation Functionality, including Code Generation:* Automated model transformations from UML to formal languages and back to

[1]Formerly named SENSORIA Development Environment.

bridge the gap between these worlds; also, generation of executable code (for example, Web Service standards like BPEL).

- *Formal Analysis Functionality:* Model checking and numerical solvers for stochastic methods based on process calculi code defined by the user or generated by model transformation.
- *Runtime Functionality:* Integration of runtime platforms, for example BPEL process engines or the Java runtime as well as runtime support for services, for example dynamic service brokering.

The functionality indicated in the previous list is implemented in various tools, some of which have been developed within the Sensoria project, some developed outside of the project. The tools have not only been developed at different sites, but are also vastly different with regard to user interface, functionality, required computing power, execution platform and programming language. However, all of the tools contribute to the development process and in many cases deliver artefacts which may serve as input to other tools.

The Service Development Environment (SDE) provides an integration of these tools through a carefully designed, lightweight integration architecture. This is achieved through the following core features:

- *Tools As Services.* The SDE itself is based on a service-oriented architecture, allowing easy integration of tools and querying the platform for available functionality. The tools hosted in the SDE are installed and handled as services.
- *A Composition Infrastructure.* As development of services is a highly individual process and may require several steps and iterations, the SDE offers a composition infrastructure which allows developers to automate commonly used workflows as an orchestration of integrated tools.
- *Eclipse Integration.* The SDE is based on the industry-standard Eclipse platform and is fully compatible with tools installed in Eclipse either with or without the SDE. This ensures maximum interoperability with a vast amount of Eclipse-based development tools and workbenches.

As with services in a SOA, tool composition in the SDE is a lightweight one, i.e., the connection between tools is not a priori fixed and including additional tools requires only minimal change to the integrated tools. Using the tool-as-a-service metaphor, tools are services, each consisting of functions which can be invoked by the user or other services. Contrary to Web services [WCL+05], user interaction is very important for some software development tools. For example, a modelling tool requires a lot of user interaction — ideally, the modelling tool runs on the computer of the user. A model checker, on the other hand, requires a lot of computing power and thus will most likely run on a dedicated server to be accessed remotely with none or only a minimal, generated UI available. Both use cases are supported in the SDE.

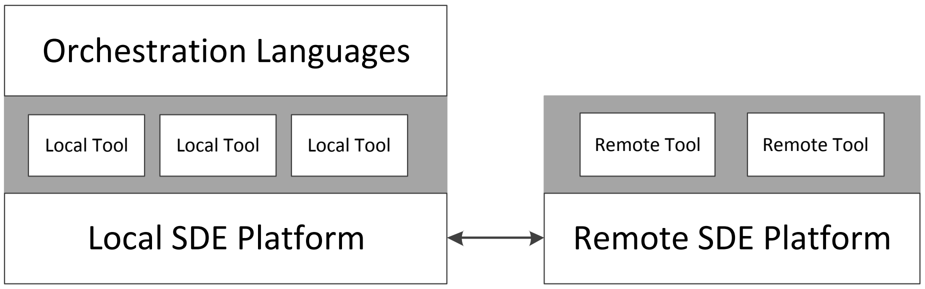


Figure 8.1: SDE: Architectural Overview

By using a SOA-based infrastructure, combining tools into more complex tool chains is straightforward, i.e. possible via dedicated orchestration languages. A typical scenario for tool composition can be found in the analysis and verification of software; for example, model checkers require a certain input format into which most source models first need to be transformed by a transformation tool; the same applies to the output. The SDE contains both a textual (JavaScript) and a graphical workflow-based orchestration language, allowing users to integrate various tools, thereby handling the data flow between these tools. Having encapsulated the integrating steps, they can be run over and over again for performing the same steps with different input and output data.

Figure 8.1 shows the architecture of the SDE. As discussed previously, the integration platform hosts a number of tools as services. Through its dedicated orchestration infrastructure, the SDE allows developers to orchestrate tools to be used in combination, with remote invocation functionality for invoking tools hosted on different machines.

8.2 The Service Development Environment

The tools and techniques developed in this thesis as well as the Sensoria project support the creation of service-oriented software by *augmenting* existing development processes and tools. A requirement for the SDE was therefore to integrate with existing tools and platforms for the development of SOA systems. For this reason, the SDE is based on the well-known Eclipse platform [Ecl10i] and its underlying, service-oriented OSGi [OSG08] framework. OSGi is based on *bundles*, which are components grouping a set of Java classes and meta-data providing among other things name, description, version, exported and imported packages of the bundle. A bundle may provide arbitrary services to the platform.

8.2.1 Architecture

The technical architecture of the SDE is shown in figure 8.2, which shows the SDE Platform as an OSGi bundle, its dependencies and dependent bundles.

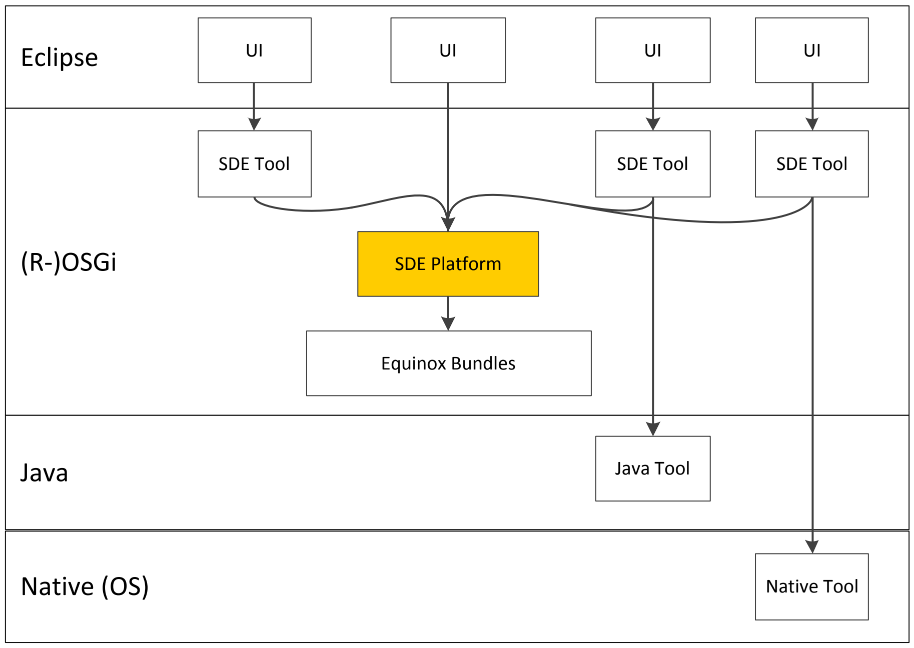
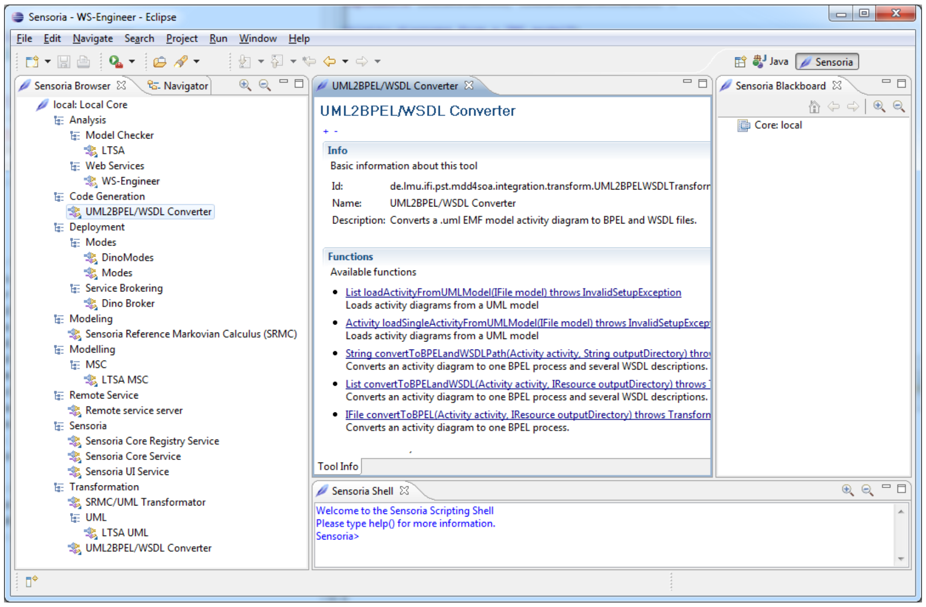


Figure 8.2: SDE: Technical Architecture

Fundamentally, all tools in the SDE are integrated as OSGi bundles which offer certain *functions* for invocation by the platform. As indicated above, the tools integrated into the SDE are vastly different, ranging from user-driven graphical modelling tools to computationally intensive analysis tools with very basic interaction mechanisms. Thus, it is not possible to define a common API for all tools. In the SDE, this problem is solved by using (declarative) OSGi services for each tool. Furthermore, the SDE allows tools to provide their own UI, but also provides a generic invocation mechanism which enables users to invoke arbitrary functions, either directly or through an orchestration. Finally, tool integration requirements should be kept low to ensure integration of as many tools as possible. The SDE re-uses OSGi and Eclipse technology and declarative service descriptions which are generated from Java annotations for a fast and straightforward integration process.

As can be seen in figure 8.2, the SDE platform and the integrated tools are based on (R-)OSGi only (or, more specifically, the Equinox implementation of OSGi [GHM$^+$05]). This means that fundamentally, tools must be implemented in Java, although they may wrap native code or remote invocations as required. Being only based on OSGi, they can be invoked completely independently from Eclipse. If they additionally choose to provide a UI, this UI is integrated into and based on the Eclipse platform, as is the UI for the SDE platform itself.

Figure 8.3 shows a screenshot of the SDE UI. On the left hand side, the *tool browser* shows installed tools available for invocation and automation. Tools



UML2BPEL/WSDL Converter

+ -

Info

Basic information about this tool

Id: de.lmu.ifi.pst.mdd4soa.integration.transform.UML2BPELWSDLTransforr

Name: UML2BPEL/WSDL Converter

Description: Converts a .uml EMF model activity diagram to BPEL and WSDL files.

Functions

Available functions

- List loadActivityFromUMLModel(IFile model) throws InvalidSetupException
  Loads activity diagrams from a UML model
- Activity loadSingleActivityFromUMLModel(IFile model) throws InvalidSetupExcep
  Loads activity diagrams from a UML model
- String convertToBPELandWSDLPath(Activity activity, String outputDirectory) thro
  Converts an activity diagram to one BPEL process and several WSDL descriptions.
- List convertToBPELandWSDL(Activity activity, IResource outputDirectory) throws
  Converts an activity diagram to one BPEL process and several WSDL descriptions.
- IFile convertToBPEL(Activity activity, IResource outputDirectory) throws Transforn
  Converts an activity diagram to one BPEL process.

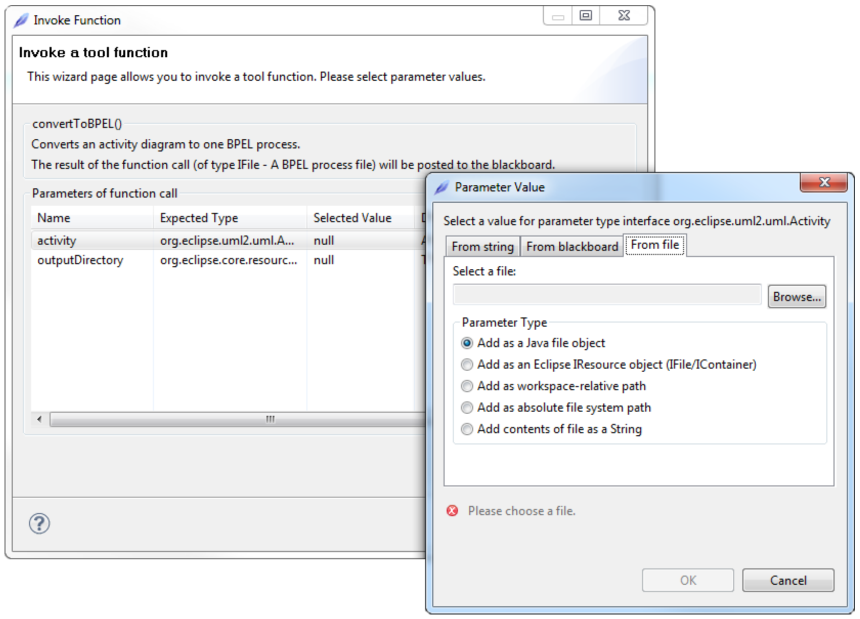
```
Welcome to the Sensoria Scripting Shell
Please type help() for more information.
Sensoria>
```

Figure 8.3: SDE: UI

are grouped by category, allowing quick access by application area. Double-clicking a tool in the browser yields more information about the tool and its functionality. This information is shown in the view in the middle: As an example, an integrated tool for model transformation (the UML2BPEL/WSDL-Converter) is shown in more detail. Each tool function displayed here can be invoked by clicking the link and providing the parameters. Finally, on the right, the *blackboard* is shown, which is a storage area where tools may place arbitrary objects for later use. Finally, at the bottom, the *shell* is displayed, which is a live JavaScript execution environment (see section 8.2.2).

As an example for a function invocation, clicking on the `convertToBPEL()` function in the UML2BPEL/WSDL tool yields the dialogs shown in figure 8.4, where the data for the two parameters (`activity` and `outputDirectory`) can be selected from various sources.

Finally, the SDE core integrates with R-OSGi [RAR07] to provide the ability to host tools for external invocation, and connect to remote SDE cores. The tools in the tool view in figure 8.3 (left), for example, are listed under the local core. Further (remote) cores may be added as required, and their tools are then listed and used in the same way as described above. Furthermore, the blackboard (right) also distinguishes between the various cores.



Invoke Function

Invoke a tool function

This wizard page allows you to invoke a tool function. Please select parameter values.

convertToBPEL()

Converts an activity diagram to one BPEL process.

The result of the function call (of type IFile - A BPEL process file) will be posted to the blackboard.

Parameters of function call

Name	Expected Type	Selected Value
activity	org.eclipse.uml2.uml.A...	null
outputDirectory	org.eclipse.core.resourc...	null

Parameter Value

Select a value for parameter type interface org.eclipse.uml2.uml.Activity

From string | From blackboard | From file

Select a file:

Parameter Type

- [✓ Add as a Java file object]
- Add as an Eclipse IResource object (IFile/IContainer)
- Add as workspace-relative path
- Add as absolute file system path
- Add contents of file as a String

Please choose a file.

Figure 8.4: SDE: Generic Invocation Wizard

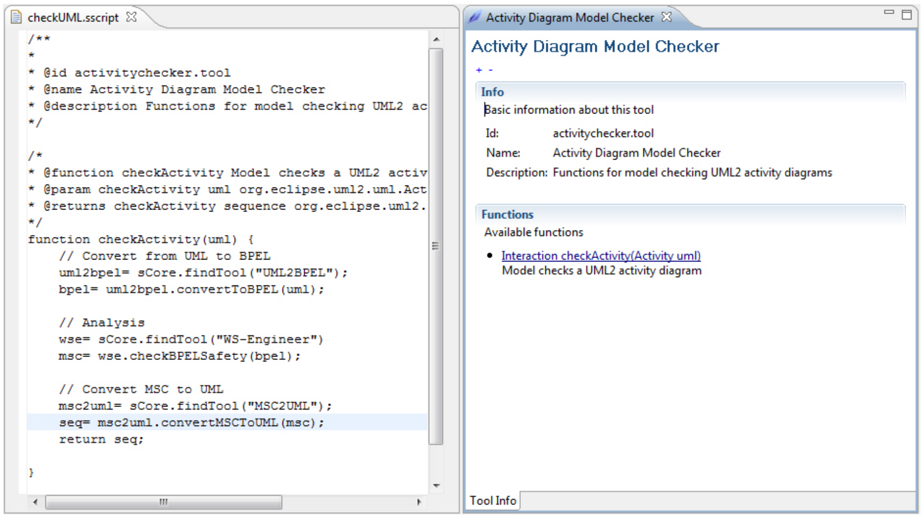
8.2.2 Composing Tools

The SDE provides the ability to compose new tools out of existing ones, which is the same concept as orchestration in SOA terminology. Creating orchestrations is possible in the SDE by using two mechanisms: A textual, JavaScript-based approach, and a graphical workflow approach.

Orchestrating with JavaScript

The ability to use tool APIs directly within JavaScript enables developers to create a workflow by invoking tool functions and passing data in-between those functions. To enable the newly created workflow to be usable as a tool in its own right, two things are required: Instead of simply creating a workflow, a JavaScript function definition is required which states a function name and parameters. As each tool, function, parameters, and return types may have descriptions and additional meta-data attached, this meta-data must be specified in some way in the JavaScript source files. Both points have been addressed in the SDE. The first is simple; function definitions are already part of the JavaScript specification. The second was solved by employing a JavaDoc-comment-style approach to meta-data specification. Tags like `@description` are used to convey meta-data information.

As an example, figure 8.5 (left) shows a script for converting UML2 activity diagrams to BPEL, then analysing them using the WS-Engineer tool, and finally converting the result back to UML2 sequence diagrams showing the error trace.



```
/**
*
* @id activitychecker.tool
* @name Activity Diagram Model Checker
* @description Functions for model checking UML2 ac
*/

/*
* @function checkActivity Model checks a UML2 activ
* @param checkActivity uml org.eclipse.uml2.uml.Act
* @returns checkActivity sequence org.eclipse.uml2.
*/
function checkActivity(uml) {
    // Convert from UML to BPEL
    uml2bpel= sCore.findTool("UML2BPEL");
    bpel= uml2bpel.convertToBPEL(uml);

    // Analysis
    wse= sCore.findTool("WS-Engineer")
    msc= wse.checkBPELSafety(bpel);

    // Convert MSC to UML
    msc2uml= sCore.findTool("MSC2UML");
    seq= msc2uml.convertMSCToUML(msc);
    return seq;

}
```

Activity Diagram Model Checker

Info
Basic information about this tool

Id:	activitychecker.tool
Name:	Activity Diagram Model Checker
Description:	Functions for model checking UML2 activity diagrams

Functions
Available functions

- Interaction checkActivity(Activity uml)
  Model checks a UML2 activity diagram

Figure 8.5: SDE: Orchestration with JavaScript

Figure 8.5 (right) shows the converted tool inside the SDE tool browser. Scripts created like this can be used on any SDE installation which has the required tools installed. No particular deployment is necessary save copying the script and registering it with the core.

For testing purposes, the SDE also contains a JavaScript live execution environment, the SDE Shell (figure 8.3), where JavaScript commands can be executed without compiling a complete script.

Graphical Orchestration

Besides the ability to use JavaScript for orchestration as indicated above, the SDE also contains the ability to orchestrate tools graphically. The syntax used is that of UML activity diagrams, although the main focus is on data flow, i.e. the flow of information from pin to pin. An activity in the diagram represents one function in the tool to be generated which has input pins (parameters) and one output pin (return type). Inside the activity, actions represent function calls to arbitrary (installed) tools. These actions have pins themselves; data flow edges model the data transfer.

As an example, consider the screenshot in figure 8.6, which shows the orchestration introduced in the previous paragraph as a graphical workflow, including the editor which supports it. The function `checkActivity(uml)` is modelled as an activity, and each call to a particular function of an installed tool is modelled as an action. On the right-hand side, the toolbar shows all available tools and the functions they provide. Once modelled, an orchestration such as the one above is converted to a Java class, compiled in-memory and installed as a tool in the SDE.

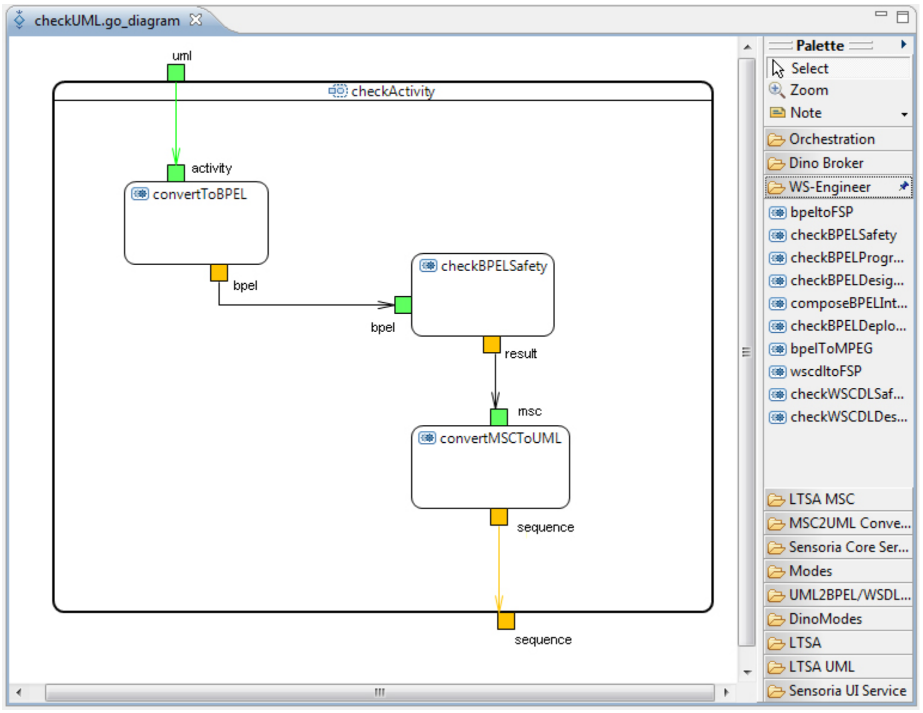


Figure 8.6: SDE: Graphical Orchestration

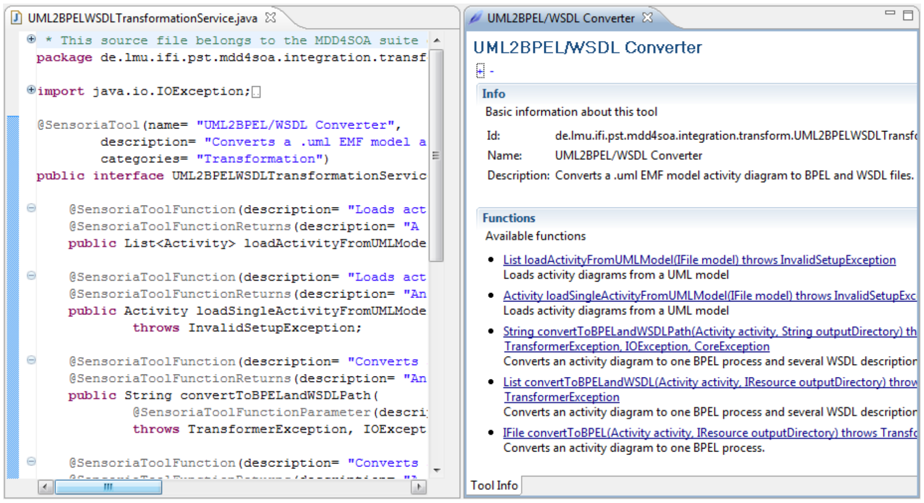
8.2.3 Extending the Platform

The SOA-based architecture of the SDE enables tool integration through a core API and an extension point for registering tools. Basically, each tool is an OSGi bundle with some published API and meta-data XML to register the tool with the SDE core. Thus, creating a facade class and registering the class with the SDE extension point enables tool functionality to be immediately available within the SDE, both for manual invocation and automation.

Tools within the SDE are loosely coupled, as they are fundamentally independent from each other and interact through their published service interfaces only. They may, of course, require other tools to be installed for them to work. This is defined in a declarative way through the Equinox extension mechanism and checked by the platform prior to tool installation.

The SDE core also contains a set of Java annotations, which enable tool developers to define their tools and functions without writing any XML. As an example, consider figure 8.7: On the left-hand side, a tool interface with SDE annotations is shown; on the right-hand side, the corresponding tool view in the SDE.

The API defined within the integration tool service bundle provides access to all installed tools. A tool may use this API to verify installation of required



```
 * This source file belongs to the MDD4SOA suite
package de.lmu.ifi.pst.mdd4soa.integration.transf

import java.io.IOException;

@SensoriaTool(name= "UML2BPEL/WSDL Converter",
        description= "Converts a .uml EMF model a
        categories= "Transformation")
public interface UML2BPELWSDLTransformationServic

    @SensoriaToolFunction(description= "Loads act
    @SensoriaToolFunctionReturns(description= "A
    public List<Activity> loadActivityFromUMLMode

    @SensoriaToolFunction(description= "Loads act
    @SensoriaToolFunctionReturns(description= "An
    public Activity loadSingleActivityFromUMLMode
            throws InvalidSetupException;

    @SensoriaToolFunction(description= "Converts
    @SensoriaToolFunctionReturns(description= "An
    public String convertToBPELandWSDLPath(
            @SensoriaToolFunctionParameter(descri
            throws TransformerException, IOExcept

    @SensoriaToolFunction(description= "Converts
```

UML2BPEL/WSDL Converter

Info

Basic information about this tool

Id:	de.lmu.ifi.pst.mdd4soa.integration.transform.UML2BPELWSDLTransf
Name:	UML2BPEL/WSDL Converter
Description:	Converts a .uml EMF model activity diagram to BPEL and WSDL files.

Functions

Available functions

- List loadActivityFromUMLModel(IFile model) throws InvalidSetupException
  Loads activity diagrams from a UML model
- Activity loadSingleActivityFromUMLModel(IFile model) throws InvalidSetupExc
  Loads activity diagrams from a UML model
- String convertToBPELandWSDLPath(Activity activity, String outputDirectory) th
  TransformerException, IOException, CoreException
  Converts an activity diagram to one BPEL process and several WSDL descriptior
- List convertToBPELandWSDL(Activity activity, IResource outputDirectory) throw
  TransformerException
  Converts an activity diagram to one BPEL process and several WSDL descriptior
- IFile convertToBPEL(Activity activity, IResource outputDirectory) throws Transfo
  Converts an activity diagram to one BPEL process.

Figure 8.7: SDE: Tool Registration

tools; search for tools based on meta-data, and invoke functionality as needed. Therefore, it serves as a discovery service which moderates between the tools. Once the connection has been made, communication between tools is done directly.

8.3 MDD4SOA Tools

This thesis has introduced several tools for aiding developers in the development of service-oriented systems. All tools presented in this thesis are open source and licensed with the Eclipse Public License (EPL) [Ope04], which is an OSI-approved license [OSI10] which guarantees source code availability and at the same time does not restrict selling, extending, and re-licensing extended versions of the code.

Figure 8.8 shows the overview figure from the introduction (page 3) again; this time with annotated tools.

The tools created as part of this thesis can be grouped into three categories. For each of these categories, a web site is available which hosts binary and source versions of the tools belonging to the category.

8.3.1 Modelling and Semantics

The first category includes tools for modelling SOA systems with UML4SOA, and for mechnically translating UML4SOA diagrams to modal input/output automata.

Firstly, this includes the UML4SOA profile itself. As discussed in chapter 3, the UML4SOA profile is available for both MagicDraw and Rational Software

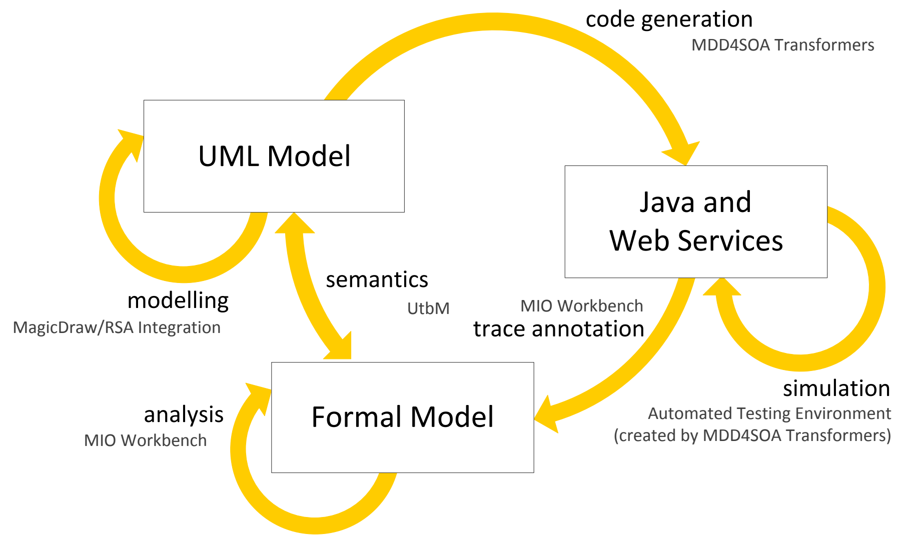


Figure 8.8: MDD4SOA Tools

Architect. Furthermore, a toolbar in form of a module is available for MagicDraw.

Secondly, the modelling category of tools also includes the UtbM tool which converts UML4SOA models to modal input/output automata, and enables back-annotation of verification results to MagicDraw UML models. The UtbM tool has been introduced in chapter 5.

The tools of this category are available from the UML4SOA website.

UML4SOA Website: `http://www.uml4soa.eu/`.

8.3.2 Verification Support

The second category consists of the Mio Workbench described in chapter 5 — an Eclipse-based verification tool and editor for modal I/O automata (MIOs). The Mio Workbench provides:

- A graphical editor for MIOs;
- Verification support for different refinement and compatibility notions; and
- Trace annotation capability.

The Mio Workbench includes all interface theories mentioned in this thesis, i.e. strong, weak, and strict-observational. Furthermore, several additional notions such as hiding and helpful environments are provided.

The Mio Workbench is available as an Eclipse plugin, and may be installed directly from the update site of the Mio Workbench website.

Mio Workbench Website: `http://www.miowb.net/`

8.3.3 Transformation and Code Generation

Transformation and code generation have been described in chapter 6. This includes the following tools:

- UML2SMM: A transformer from an UML4SOA model created with a modelling tool such as MagicDraw, Rational Software Architect, or the Eclipse UML2 tools to an instance of the SMM.
- SMM2Java: A transformer from the SMM to Java. Note that this tool optionally takes UML as input, performing the UML2SMM step as well.
- SMM2WS: A transformer from the SMM to the Web Service standards family, which includes the creation of BPEL, WSDL, and XSD files. As in the case of SMM2Java, this tool optionally takes UML as input, performing the UML2SMM step as well.

Each of the transformers is available as an Eclipse plugin, and may be installed directly from the update site of the MDD4SOA Website.

MDD4SOA Website: `http://www.mdd4soa.eu/`.

8.4 Other Integrated Tools

This chapter lists other tools which have been integrated into the SDE platform, sorted by integrated category.

8.4.1 Modelling

MagicDraw

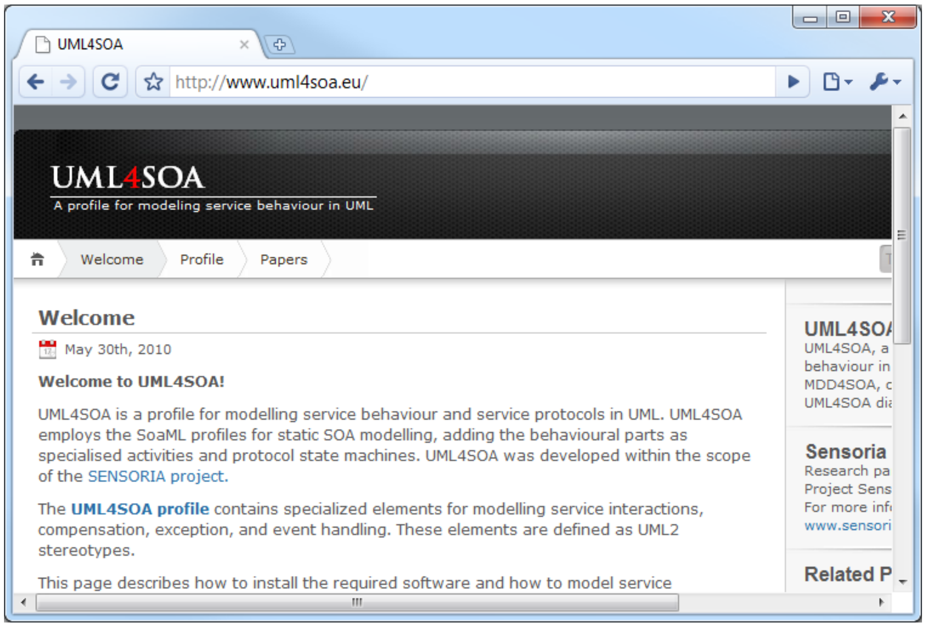
MagicDraw [NoM10] is a platform-independent UML modeller with profile support for UML2.

`http://www.magicdraw.com/`

Rational Software Architect

Rational Software Architect [IBM09] is a UML modelling tool which supports UML2 profiles and is built on the Eclipse platform.

`http://www.ibm.com/software/awdtools/architect/swarchitect/`



Welcome

May 30th, 2010

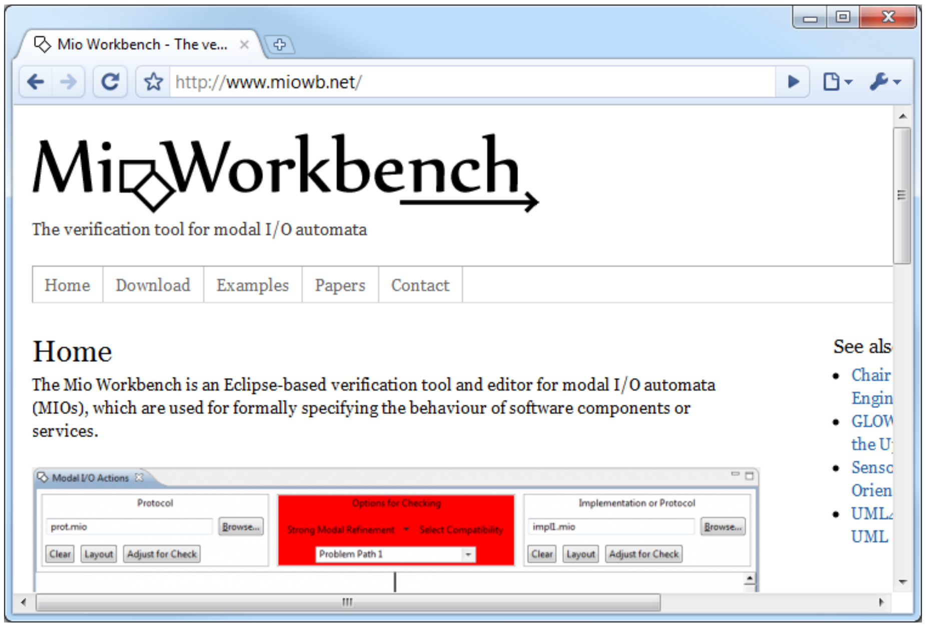
Welcome to UML4SOA!

UML4SOA is a profile for modelling service behaviour and service protocols in UML. UML4SOA employs the SoaML profiles for static SOA modelling, adding the behavioural parts as specialised activities and protocol state machines. UML4SOA was developed within the scope of the SENSORIA project.

The **UML4SOA profile** contains specialized elements for modelling service interactions, compensation, exception, and event handling. These elements are defined as UML2 stereotypes.

This page describes how to install the required software and how to model service

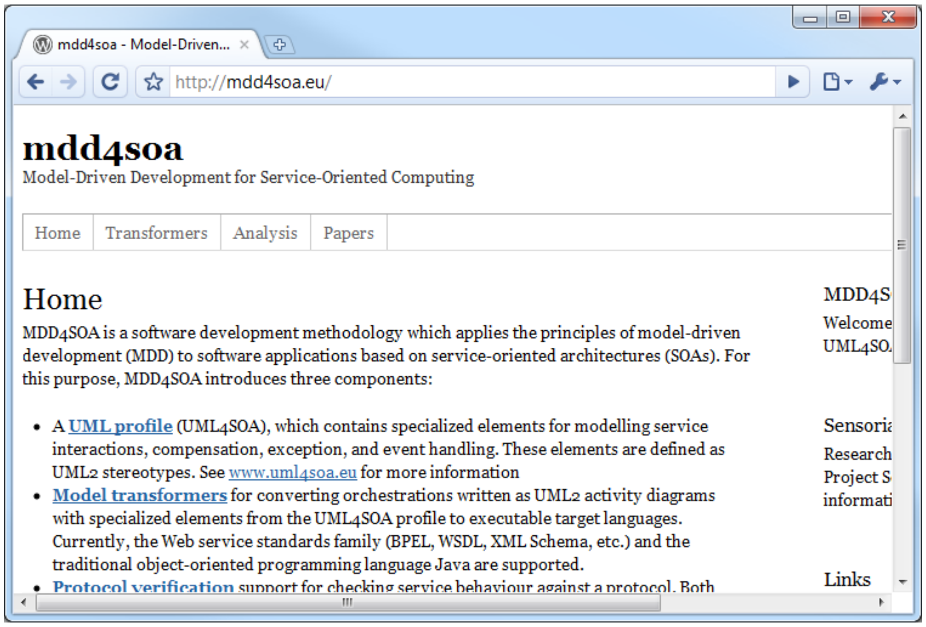
Figure 8.9: UML4SOA Website



Home

The Mio Workbench is an Eclipse-based verification tool and editor for modal I/O automata (MIOs), which are used for formally specifying the behaviour of software components or services.

Figure 8.10: Mio Workbench Website



mdd4soa

Model-Driven Development for Service-Oriented Computing

Home | Transformers | Analysis | Papers

Home

MDD4SOA is a software development methodology which applies the principles of model-driven development (MDD) to software applications based on service-oriented architectures (SOAs). For this purpose, MDD4SOA introduces three components:

- A UML profile (UML4SOA), which contains specialized elements for modelling service interactions, compensation, exception, and event handling. These elements are defined as UML2 stereotypes. See www.uml4soa.eu for more information
- Model transformers for converting orchestrations written as UML2 activity diagrams with specialized elements from the UML4SOA profile to executable target languages. Currently, the Web service standards family (BPEL, WSDL, XML Schema, etc.) and the traditional object-oriented programming language Java are supported.
- Protocol verification support for checking service behaviour against a protocol. Both

MDD4S

Welcome

UML4SO

Sensori

Research

Project S

informati

Links

Figure 8.11: MDD4SOA Website

8.4.2 Transformation

Hugo/RT

Hugo/RT [SKM01] is a UML model translator for model checking, theorem proving, and code generation: A UML model containing active classes with state machines, collaborations, interactions, and OCL constraints can be translated into the system language of the real-time model checker UPPAAL, the on-the-fly model checker SPIN, the system language of the theorem prover KIV, and into Java and SystemC code.

`http://www.pst.informatik.uni-muenchen.de/projekte/hugo/`

VIATRA2

The main objective of the VIATRA2 (VIsual Automated model TRAnsformations) framework [BV06] is to provide general-purpose support for the entire life-cycle of engineering model transformations including the specification, design, execution, validation and maintenance of transformations within and between various modelling languages and domains.

`http://wiki.eclipse.org/VIATRA2`

SOA2WSDL-Transformation

The SOA2WSDL [ÁvédVG06] transformation, written in VIATRA2, takes high level UML models and produces WSDL output.

`http://viatra.inf.mit.bme.hu/`

SRMC/UML Bridge

The SRMC/UML bridge [TG08] offers facilities for meta-model transformation. It translates a subset of UML2 models (Interactions and State Machines) into an SRMC description for performance evaluation. Results are reflected back into the UML model.

`http://groups.inf.ed.ac.uk/srmc/`

Modes Parser and Browser

The Modes Parser and Browser [FMRU08] is a WS-Engineer plug-in to parse and extract broker requirements from UML2 Modes Models.

`http://www.doc.ic.ac.uk/ltsa/eclipse/wsengineer`

8.4.3 Analysis

LTSA

LTSA [MK99] is a verification tool for concurrent systems. It checks that the specification of a concurrent system satisfies the properties required of its behaviour. In addition, LTSA supports specification animation to facilitate interactive exploration of system behaviour.

`http://www.doc.ic.ac.uk/ltsa/`

WS-Engineer

The LTSA WS-Engineer plug-in [FUMK06] is an extension to the LTSA Eclipse Plug-in which allows service models to be described by translation of the service process descriptions, and can be used to perform model-based verification of Web Service compositions.

`http://www.doc.ic.ac.uk/ltsa/eclipse/wsengineer/`

SRMC/PEPA

The SRMC (SENSORIA Reference Markovian Calculus) tool [CGT09a] provides support for SRMC, an extension to PEPA. It covers steady-state analysis of the underlying Markov chain of SRMC descriptions.

`http://groups.inf.ed.ac.uk/srmc/`

SPIN

Spin [Hol97] is an open-source software tool used for the formal verification of distributed software systems.

`http://spinroot.com/`

UPPAAL

Uppaal [BLL$^+$95] is an integrated tool environment for modelling, validation and verification of real-time systems modelled as networks of timed automata, extended with data types (bounded integers, arrays, etc.).

`http://www.uppaal.com/`

CMC / UMC

CMC and UMC [tBMG09] are model checkers and analysers for systems defined by interacting UML state charts. Both allow on-the-fly model checking of abstract behavioural properties in the Socl branching-time state-action based, parametric temporal logic.

`http://fmt.isti.cnr.it/cmc/,http://fmt.isti.cnr.it/umc/`

LySa tool

LySa [Buc05] is a static analyser for security protocols defined in the LYSA process calculus. The tool provides a LYSA editor to assist users in the modelling of protocols. Given a LYSA model the analyser will verify properties related to secrecy and authentication.

`http://www2.imm.dtu.dk/cs_LySa/lysatool/`

8.4.4 Deployment and Runtime

UML2AXIS Transformation

The UML2AXIS [ÁvédVG06] transformation, written in VIATRA2, takes UML models with specifications of non-functional attributes for reliable messaging, and produces Web Service code based on the Apache Axis Java library.

`http://viatra.inf.mit.bme.hu/`

Dino Broker

The Dino Broker [FMRU08] provides dynamic runtime discovery of services which are described in OWL and WSDL documents, thus enabling developers to bind services which correspond to specific criteria.

`http://www.cs.ucl.ac.uk/research/dino/`

8.5 Tool Applications

The tools listed in the previous chapter can be combined in various ways to achieve different transformations and analyses. Figure 8.12 lists, non-exhaustively, the links between the tools.

As examples, we provide three scenarios with different tools to give some insights into how tools may be chained together in the SDE. In the following sections, we use four paragraphs to describe each scenario:

- *Use Case* describes when and why to use a certain tool chain.
- In *Tools Involved*, we list the tools required to perform the functionality of the scenario.
- *Data Flow* shows the individual steps to be executed in the tool chain.
- Finally, *Results* describes the consequences and benefits of using the scenario.

The tool chains may be realised manually, i.e. with the user performing one step after another and storing the intermediate objects on disk or on the blackboard, or automatically by employing JavaScript or the graphical orchestration mechanism.

8.5.1 Analysis and Code Generation of Services

Use Case

This use case describes, in short, the techniques and tools provided by this thesis, i.e. a model-driven development approach with analysis support for services and service orchestrations. A customised UML2 profile for modelling SOAs (UML4-SOA) is used to create a model of the target SOA system. Besides modelling the behaviour itself, a behavioural protocol can help to assess the external behaviour of the orchestration and used to verify the actual implementation. Once a service orchestration has been verified, it needs to be transformed to code in target languages like BPEL or Java to deploy it for execution.

Tools Involved

This tool chain includes a UML modeller with profile support, like MagicDraw or Rational Software Architect. The Mio Workbench is used to report on protocol violations. Finally, the model transformers of MDD4SOA are used to transform the UML specifications to code in executable languages (for example, BPEL and WSDL) for deployment.

Data Flow

The chain starts with the user who employs a UML modeller to design both the orchestration implementation and the service protocol. The resulting diagrams

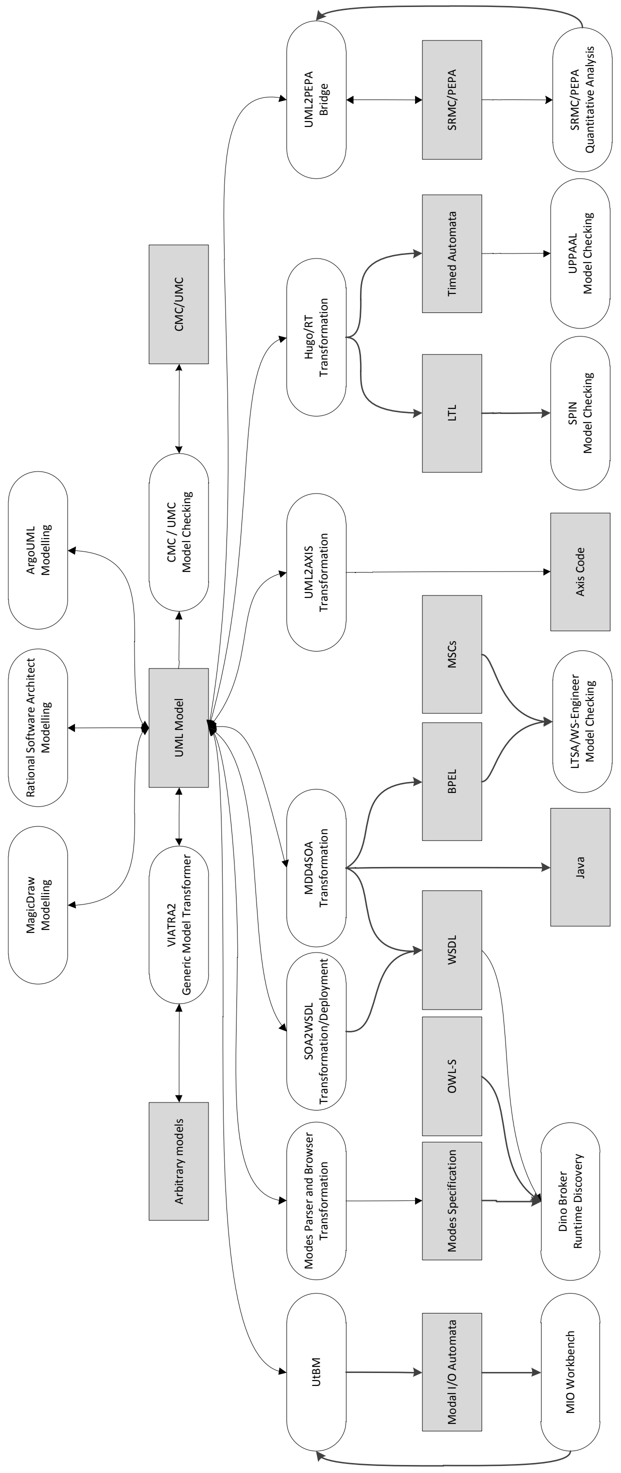


Figure 8.12: SDE: Available Tool Chains

are saved as documents in the XMI format. These files can then be used by the UtBM to convert to MIOs, and checked using the Mio Workbench, which either reports no protocol violations or creates a violation trace which is back-annotated to UML by using UtbM. This process is repeated until the process is error-free. Finally, the UML2 models are read by the MDD4SOA Transformers, which generate the appropriate target code, depending on which language has been selected by the user.

Results

Chaining tools together in this fashion enables the developer to quickly react to changes in requirements, as the chain can be run automatically whenever a change has occurred, either informing the user of newly introduced problems in the protocol or, if the protocol is valid, creating the new implementation in the selected target language.

8.5.2 Qualitative and Quantitative Analysis

Use Case

Service-oriented software systems are commonly distributed — they make use of a network to combine various individual software components to work in coordination to reach a higher-level goal. In general, a SOA system contains many different threads of execution, which run in parallel and interact with one another in nontrivial ways. This poses a difficult problem to software designers, as the interaction of such threads needs to be analysed in order to ensure that no undesirable effects occur. Furthermore, it is not always clear how the system time is spent during runtime. Therefore, mechanical checkers are needed to verify whether a certain implementation is free from conditions such as deadlocks, and secondly for assessing the runtime characteristics of the overall system.

Tools Involved

Again, we employ UML modellers like Rational Software Architect or MagicDraw for the modelling of a service-oriented system written in UML. Based on these models, quantitative analysis as well as qualitative analysis is then performed by the SRMC tool and the WS-Engineer tool, respectively. While the former is able to deal with UML directly, the latter requires the BPEL format as input, so another tool is used (one of the MDD4SOA transformers) for converting between UML and BPEL.

Data Flow

The chain starts with the user who employs a UML modeller to design a model of communicating systems in UML2. The resulting model, in the format of an UML2 XMI file, can be read directly by the SRMC tool to report on the distribution of time spent in the various states of the process. Using the MDD4SOA transformers, the UML2 model is converted to BPEL to serve as input for WS-Engineer, which is used to verify the required properties (for example, freeness from dead-locks). Finally, the result of the analysis is shown to the user: The quantitative analysis can be directly annotated to the original UML model (or output as graphs), the qualitative analysis — if resulting in an error trace — is shown as Message Sequence Charts (MSCs).

Results

This tool chain provides the user with a "one-click" verification of the model — instead of requiring the user, as is common in many verification tools, to activate a translation of service implementations, feed the translation through a model parser, compile the model, and invoke a verify option on the model checker. All these single steps are handled by the tool chain and the script used to combine the two different analysers. Thus, checking becomes less of a hassle and will be executed more often, resulting in higher-quality systems.

8.5.3 Modes-Based Dynamic Runtime Discovery

Use Case

One of the promises of the Service-Oriented Architecture is the ability to quickly react to changes, for example — on the business level — a change of a business partner, or — on a technical level — network connection problems or server overload. To deal with these problems, the concept of dynamic service discovery and binding has been introduced, which enables developers to specify, on an abstract level, the properties and constraints required of certain services needed by an orchestration. Specification of such properties, the criteria of when to change the service to be used (specified by "modes"), and testing of the resulting runtime behaviour are non-trivial issues, and tool support is needed to make such approaches practical.

Tools Involved

The main focus of this tool chain lies on testing of dynamic service discovery, hence the most important tool is the Dino Broker used for service discovery. Serving input to Dino is the Modes Parser and Browser Tool which handles translation of modes from the UML2 models. Dino also requires WSDL and OWL documents for service specification which can, in part, be generated by the VIATRA2 SOA2WSDL transformation tool. Again, the initial mode specification is done in UML2, for which a UML2 modeller is required.

Data Flow

The chain starts with the user who employs a UML modeller to design a model of a SOA system enhanced with mode specifications and the required constraints on services. The Modes Parser and Browser Tool is then used to convert these specifications to input for the Dino Broker. In parallel, the services to be discovered are deployed to the Dino runtime, either from pre-existing OWL/WSDL specifications or from those generated by the SOA2WSDL transformation. Finally, the developer can employ the Dino Broker front-end which is available through the SDE to test-drive the service discovery, and once satisfied, use the generated documents for the final implementation.

Results

The ability to generate input for Dino from UML2 and test-driving the discovery right from within the development environment greatly speeds up the process of finding the right mode and constraint specifications. Automation allows writing test cases for the complete process, thus the user may change the specifications at the beginning of the chain and verify the output stemming from an actual discovery run with the Dino Broker, saving time and effort in debugging.

8.6 Related Work

There are several distinguishing features of the SDE which sets it apart from standard development environments. The main aspect is the idea of SOA-based tooling, i.e. regarding the development environment as a SOA itself, which leads to the concept of tools as services, a discovery mechanism, and orchestration support. Furthermore, the integrated tools are at least partly graphical and interactive, requiring fast response time for developers and thus, a local installation. This effectively prevents the use of solutions based only on Web Services.

In the following, integration platforms with similar aims are discussed and compared to the SDE. One exception is Eclipse, which is discussed first as a baseline for comparison.

The first question to answer is why the basis of the SDE, i.e. Eclipse itself, is not sufficient for tool integration purposes. Eclipse, as a platform, indeed provides many of the features which are required for tool integrations, such as the ability to extend the platform at certain points, using update sites for discovering and installing software, and the workbench concept of action contributions. However, the SDE provides two important aspects which are missing in Eclipse.

First, this is the concept of providing arbitrary, machine-invokable services and their functions — the Eclipse platform only allows contributions to existing extension points, but not in a generic way. Second, and this feature requires the first, is the concept of service composition or orchestration without writing custom code: The SDE contains both a textual and a graphical orchestration mechanism for combining tool functionality, which, again, is not provided by Eclipse.

The following discussion is restricted to tools which are based on a SOA-like structure, thus enabling tools to add generic, machine-invokable functionality in a uniform way.

The EU funded project SeCSE [SeC10] provides a middleware, the SeCSE IF (Integration Framework) and a corresponding development environment, also called SDE (SeCSE Development Environment). The integration framework is based on Web Service- and Java technology. Development and runtime are integrated, i.e. the IDE can be used for monitoring and runtime changes as well. The development environment is provided as a web-only custom solution and is thus limited with regard to modelling tool support.

The jETI framework [MNS05] is a redesign of the Electronic Tools Integration platform (ETI) in Java and uses Web Services. It is intended for remote tool integration of verification tools and allows integration, organisation, and execution of remote functionalities. jETI also provides a client, which can be used for the coordination of the various tools, either stand-alone or as an Eclipse plug-in. Tools within jETI are represented as Web Services, which means that they are executed on the remote site and thus are again limited regarding the user interface options. While it is possible to start a remote tool with its user interface redirected to the computer where the user of the tool is located, it requires a high bandwidth connection in case of a graphical user interface.

PWeb [BBFR04] is a tool integration platform based on Web Services developed in the EU project Profundis [PRO05]. Its goal is the integration of semantic-based verification toolkits which are accessible as Web Services themselves. PWeb comes with a directory service which offers the primitives to publish a service and to query for a service. In principle, the discovery service can be distributed as it can contain references to other discovery services.

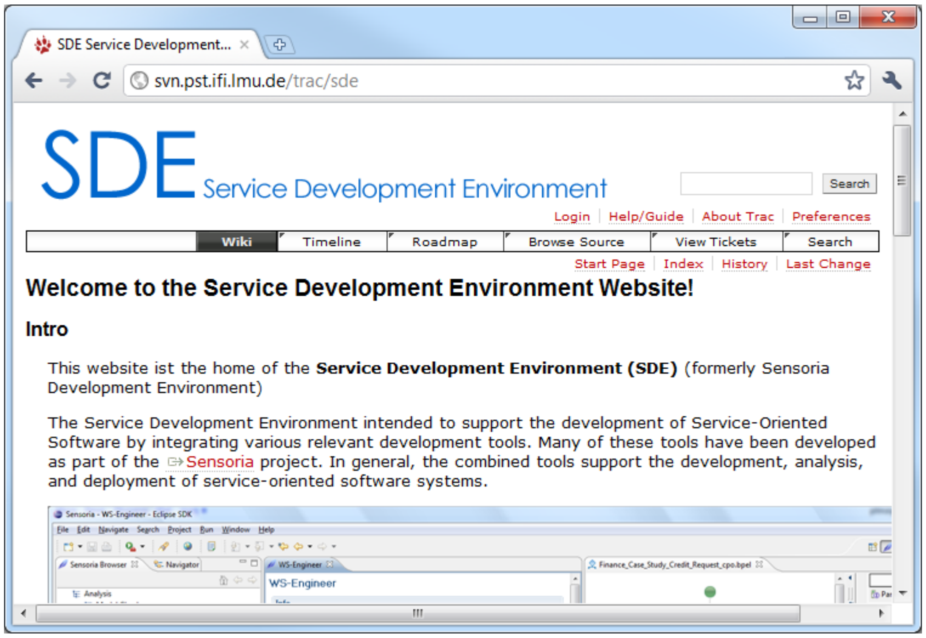
A main focus of PWeb is the possibility to coordinate the various tools by using Python as the orchestration language. PWeb focuses only on Web Services, again rendering it difficult to integrate modelling tools which have their own user interface. Furthermore, PWeb itself only provides a rudimentary Web interface as the user interface.

Finally, an interesting approach to tool integration is the FUJABA tool [BGT$^+$04], which places special emphasis on the data required by individual tools, and provides support for meta-model level data integration. This approach is orthogonal to the principles behind the SDE, as it does not offer SOA-based tool integration itself.

8.7 Summary

In this chapter, we have discussed the need for, requirements of, design, and usage of a tool integration platform for the development of service-oriented software systems. Based on a service-oriented architecture itself, the Service Development Environment (SDE) contains tools for modelling and analysing service artefacts as well as generating code and supporting services at runtime, allows remote invocation of tool functionality, and enables composition of tools by a textual and graphical orchestration mechanism.

We believe that thinking of individual development tools as services and including SOA features like self-describing services, remote invocation, and or-


SDE Service Development...
svn.pst.ifi.lmu.de/trac/sde
SDE Service Development Environment
Search
Login | Help/Guide | About Trac | Preferences
Wiki | Timeline | Roadmap | Browse Source | View Tickets | Search
Start Page | Index | History | Last Change

Welcome to the Service Development Environment Website!

Intro

This website ist the home of the **Service Development Environment (SDE)** (formerly Sensoria Development Environment)

The Service Development Environment intended to support the development of Service-Oriented Software by integrating various relevant development tools. Many of these tools have been developed as part of the Sensoria project. In general, the combined tools support the development, analysis, and deployment of service-oriented software systems.

Figure 8.13: SDE Website

chestration into a tooling environment greatly extends the applicability of the integrated tools.

This chapter has also listed tools integrated into the SDE, and has discussed three end-to-end examples of development workflows created with the orchestration functionalities available in the SDE.

The SDE is available for download on `http://svn.pst.ifi.lmu.de/trac/sde` (see figure 8.13). This page also includes a tutorial for tool integration and videos demonstrating the SDE in action.

Chapter 9

Conclusion

The topic of this thesis has been the integration of Model-Driven Development (MDD) and the paradigm of Service-Oriented Architectures (SOAs). This has led to the development of a comprehensive approach for *model-driven development of service-oriented systems*, which we have termed MDD4SOA. Both the methodological as well as tool-based integration aspects have been considered.

The complete MDD4SOA approach is shown again in figure 9.1.

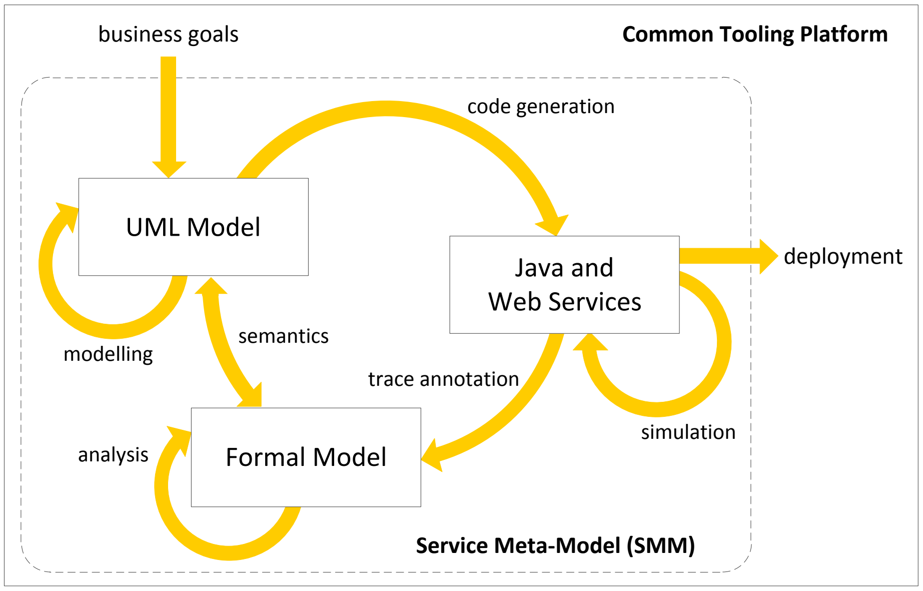


Figure 9.1: The MDD4SOA Approach

We have addressed the area of *modelling* with the high-level, domain-specific service modelling language UML4SOA. Second, we have provided a rigorous formal *semantics* and *analysis* support for this language. The area of *code generation* provides model transformers and code emitters for generating executable code from UML4SOA. Finally, our *simulation* and *trace annotation* approach allows executing the generated code and comparing the runtime behaviour with the formal semantics. All components share a common basis, the *Service Meta-Model (SMM)*, which is shown as a dashed rounded rectangle.

Furthermore, we have developed a common *tool integration platform* (enclosing component) which integrates all tools from this thesis as well as external ones contributing to the overall model-driven development process for software systems based on service-oriented architectures.

9.1 Contributions

In more detail, the contributions of this thesis are the following:

- *UML Modelling.* We have contributed the UML4SOA profile for behavioural modelling of services and service protocols to the Unified Modeling Language (UML). During the development of this profile, care has been taken to create a minimal extension, i.e. to use as many of the concepts already present in the UML and only add concepts which add to the clarity and ease of use of the profile. Furthermore, the profile is based on the upcoming OMG standard SoaML for structural modelling of SOAs, thus increasing applicability and usefulness for practitioners.

- *Semantics and Analysis.* A second contribution of this thesis has been a rigorous formal semantics for UML4SOA (via the Service Meta-Model), thus enabling formal analysis of behavioural service models. The thesis has shown how to use *interface theories* for protocol verification of SOA implementations, and has contributed the high-level, protocol-focused *strict-observational* interface theory for early verification of SOA designs.

- *Code Generation.* Model-driven development comes with the promise of being able to use models for more than just communication between developers and the ability for analysis. This thesis has thus investigated model transformations from UML4SOA models (again via the SMM) to executable target languages. Two end-to-end transformations have been provided; the first targeting the Web Service standards family languages, the second targeting Java.

- *Simulation and Tracing.* We have provided a simulation and tracing approach which enables direct execution of the generated Java implementation and thus the UML4SOA model. Furthermore, traces from such executions can be compared with the transition systems of the semantics, yielding additional information for developers as well as a validation of the generated code.

- *Integrated Tooling.* Finally, we have developed the Service Development Environment (SDE), an Eclipse-based, service-oriented integration platform for development tools targeted at SOA systems. The SDE integrates all tools provided in this thesis and additional tools from both academia and industry, and includes the ability to combine tools, thereby creating development workflows tailored to the problem at hand.

As mentioned above, the main components of the MDD4SOA approach are based on a common Service Meta-Model (SMM). The SMM employs parts of the UML, the UML4SOA profile, and the SoaML profile as its concrete syntax (modelling component), and modal input/output automata (semantics and analysis component) for the definition of its semantics. Furthermore, SMM models can be translated to executable code (code generation) and executed (simulation and tracing).

We have shown the practicability of the results presented in this thesis by application to the five case studies of the Sensoria project [WBC+09]. One of the case studies has been selected for a more thorough investigation: The eUniversity case study has accompanied us throughout the chapters of this thesis, demonstrating a complete application of the MDD4SOA process from modelling via analysis to code generation.

9.2 Discussion

This thesis has tackled various problems in its aim of integrating the domain of model-driven development and the architectural design of service-oriented software systems. In the following discussion, we highlight three interesting areas we believe to be important both for evaluating the results of this thesis and as a basis for future work.

9.2.1 Model-Driven Development of SOAs

One of the initial assumptions of this thesis has been the suitability and usefulness of model-driven development for service-oriented software systems [Fra03]. We believe that the results of this thesis support this claim.

Firstly, SOA artefacts reside on a higher level than traditional object-oriented models created in UML or cast in Java. This means that modelling and programming of SOA systems in these languages is cumbersome, as the UML4SOA chapter and the transformation to Java have shown. Creating a new level of abstraction on which to specify SOA models has the benefit of enabling developers to use services and service-related concepts without any technical overhead. The UML4SOA modelling language presented in this thesis provides this level of abstraction. Furthermore, the rigorous formal semantics and analysis techniques introduced in chapter 5 provide an early feedback mechanism on the same level as the model, which is a direct benefit of the MDD approach to SOA.

Secondly, the code generation part has shown that with regard to Web Services, a single SOA UML model necessitates artefacts in six different languages with additional requirements on the runtime platform. Clearly, availability of a unified model greatly increases usability for developers of SOA systems: Starting from a UML4SOA model, the code generators provided in chapter 6 allow developers to keep all relevant information in one model, generating the runtime artefacts as required, which is again a benefit of the MDD4SOA approach.

With regard to Java, we have shown that besides the generation of the actual implementation, a further way of exploiting the models is the generation of an automated testing environment which can be used for executing the generated code; traces from such executions can be compared against the formal semantics. This is another benefit of using models in the development of software systems.

9.2.2 The Purpose of Models

While studying how to design models and modelling languages to enable MDD4-SOA, three different usage scenarios had to be considered, each with different requirements on model design. The first scenario was using models for communication between software developers; the second was employing (behavioural) SOA models for formal analysis, and the third was using models as the basis for the generation of executable code.

Graphical models in the classic sense and as initially introduced by the UML have been designed for *communication* between software developers. A model intended for this purpose has the requirements of being readable, intuitive, understandable, and maintainable. In fact, such systems are sometimes not even created with tool support, but drawn on white boards or sheets of paper [Cra04]. A benefit of this approach is the ability to combine it with discussion techniques such as brainstorming, in which the model might even be enriched with artefacts from non-software domains.

These requirements differ greatly from the second usage scenario, which was the use of formal methods for the analysis of a model. Here, the requirements are based on the need to build a straightforward semantics for the model, i.e. creating a clear mapping to the formal foundation. The smaller the modelling language (and the model), the simpler the mapping. At the very least, it should be possible to link the formal artefacts closely to the original model, while at the same time keeping them as minimal as possible to enable analysis.

Finally, the usage scenario of code generation introduced a third set of requirements. To enable code generation, a model must be complete in the sense that each model element must be fully specified, i.e. there may be no opaque behaviours or unresolvable ambiguities. Furthermore, the model must be constructed in a way which allows the translation into a target executable language.

In this thesis, we have attempted to create a unified modelling approach which addresses all three usage scenarios and their requirements. We believe that only unified models can reap the full benefit of model-driven development, which necessitates finding the right balance between formality and usability.

As we have seen in the UML4SOA chapter, we provide two versions of UML-4SOA. The first version is intended for communication: UML4SOA/Open is not formally precise, but expressive and ideal for human communication during the initial, unstructured attempts to finding a system design. As soon as the architecture matures, models created with UML4SOA/Open can be refined to UML4SOA/Strict models. While the latter are restricted in available model elements, they are still useful for communication in a more settled phase of the software development process.

The advantage of UML4SOA/Strict models is their suitability for both formal analysis and code generation. Regarding analysis, chapter 5 has shown a semantics for UML4SOA based on the domain of modal input/output transition systems, which map closely to the actions and transitions of UML4SOA and even enable a tool-supported mapping of analysis results back to UML4SOA models. Regarding code generation, chapter 6 has shown how a translation of UML4SOA models to two industry-standard executable target languages can be carried out; in the simulation and tracing section (chapter 7), we have also seen how to generate additional external code out of the models for simulation purposes.

9.2.3 SOA Tooling

The development of an integration platform for SOA development tools has initially been targeted at the tools for formal analysis developed in Sensoria. However, the inclusion of other tools such as modelling and transformation support has proven to greatly enhance the benefit of the platform for model-driven development of SOAs. Of particular importance in this context are two features of the Service Development Environment: The first is the ability to integrate tools without introduction of a major overhead through the service-based architecture. The second is the composition support within the platform, which allows the combination of several tools to perform as a whole: In particular in the case of formal analysis, several tools need to work together to produce an output which is relevant to the software engineer, for example through generation of graphical representations of analysis results or re-annotation of violation traces on the original model.

A difficult aspect of tool integration is data: Tools must be aware of common data formats to enable sharing of artefacts. While basing tool in- and output on standardised (EMF) meta-models as done here can alleviate some of the problematic aspects of data integration, this topic remains an open area to be aware of when designing and implementing integration platforms.

9.3 Future Work

The domain of model-driven development of service-oriented systems offers several areas in which further work is required.

As has been discussed in chapter 3, modelling support for SOA architectures based on UML is still in its infancy, with the first OMG standard relating to service architectures (SoaML) just having entered beta state. SoaML and UML-4SOA now provide a solution for modelling both the static and structural aspects of SOAs in UML; furthermore, the profiles introduced in Sensoria address several additional aspects of SOAs. Still, there is more to be done. A first problem to be solved is finding a suitable way for modelling dynamic services in SOA UML models, which includes service endpoint manipulation and dynamic discovery. A further problem to be addressed are security concerns, which play a major role in open SOA platforms; adding support for modelling these concerns to UML and the model-driven process will greatly increase the value of UML models in security-critical applications. Finally, integration of user interaction with SOA behaviour is another interesting area.

As already shown in the related work section of chapter 5, several formal methods and tools are available for the analysis and verification of SOA and UML4SOA models. Nevertheless, there is opportunity for additional work here. A first important aspect in SOA applications is data. Taking this domain into consideration during verification processes is a very difficult problem, but will certainly lead to additional benefits for developers of SOA systems. Second, as mentioned above, the introduction of security aspects into SOA models allows their exploitation in formal analysis. Finally, we believe that quantitative methods are an important area of research with regard to large-scale SOA systems, and SOA modelling can benefit from the performance predictions and simulations provided by these methods.

Lastly, we believe tool support to be of paramount importance in the MDD4SOA domain. The past decade has shown tremendous advances in integrated development environments for traditional programming languages such as Java, and it is important to transfer these results to model-driven development of service-oriented architectures; in particular with the integration of formal analysis. The Service Development Environment (SDE) has shown how a SOA-based platform can be employed to integrate different tools from modelling via analysis to code generation. Maturing such platforms is an important task to be done in the future.

Finally, although the Sensoria project has included three end-to-end case studies starting from modelling via analysis and finally code generation, there is plenty of opportunity for further validation of the MDD4SOA approach in everyday industry life.

9.4 Final Words

This thesis has provided an end-to-end example of model-driven development for service-oriented systems. In our opinion, both MDD and SOAs will have a lasting impact on software engineering, and we believe that their integration is key to the development of next-generation computing systems. Our contribution to this area consists of the high-level modelling language UML4SOA, a formal semantics and analysis method for UML4SOA models, model transformations from UML to the Web Service standards family and the Java programming language with execution support, and integrated tools for the complete process of model-driven development of service-oriented systems.

Bibliography

[ACKM03] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures, and Applications.* Springer, Berlin, 2003.

[Act10] Active Endpoints. The ActiveBPEL Engine, 2010. http://www.activevos.com/community-open-source.php.

[AdAdS$^+$06] B. Thomas Adler, Luca de Alfaro, Leandro Dias da Silva, Marco Faella, Axel Legay, Vishwanath Raman, and Pritam Roy. Ticc: A Tool for Interface Compatibility and Composition. In Thomas Ball and Robert B. Jones, editors, *18th Int. Conf. Computer Aided Verification, CAV 2006*, volume 4144 of *LNCS*, pages 59–62. Springer, 2006.

[AGGI03] Jim Amsden, Tracy Gardner, Catherine Griffin, and Sridhar Iyengar. Draft UML 1.4 Profile for Automated Business Processes with a mapping to BPEL 1.0. Technical report, International Business Machines Corporation (IBM), 2003. http://www.ibm.com/developerworks/rational/library/content/04April/3103/3103_UMLProfileForBusinessProcesses1.1.pdf.

[AKCF$^+$08] Ashok Argent-Katwala, Allan Clark, Howard Foster, Stephen Gilmore, Philip Mayer, and Mirco Tribastone. Safety and response-time analysis of an automotive accident assistance service. In Margaria and Steffen [MS08], pages 191–205.

[Amb05] Thomas Ambühler. UML 2.0 Profile for WS-BPEL with Mapping to WS-BPEL. Technical report, Universität Stuttgart, 2005.

[And94] Andre Arnold. *Finite Transition Systems: Semantics of Communicating Systems.* Prentice Hall, June 1994.

[Apa10a] Apache Software Foundation. Axis: The Apache Web Services Project, 2010. http://ws.apache.org/axis/.

[Apa10b] Apache Software Foundation. ODE: The Apache Orchestration Director Engine, 2010. http://ode.apache.org/.

[Apa10c] Apache Software Foundation. Tomcat: A Java Servlet Container, 2010. http://tomcat.apache.org/.

[ÁvédVG06] János Ávéd, Dániel Varró, and László Gönczy. Model-based deployment of web services to standards-compliant middleware. In Miguel Baptista Nunes Pedro Isaias, editor, *Proc. of the Iadis International Conference on WWW/Internet 2006(ICWI2006)*. Iadis Press, 2006.

[BBFR04] Michael Baldamus, Jesper Bengtson, Gian Luigi Ferrari, and Roberto Raggi. Web services as a new approach to distributing and coordinating semantics-based verification toolkits. *Electr. Notes Theor. Comput. Sci.*, 105:11–20, 2004.

[BGT$^+$04] Sven Burmester, Holger Giese, Jörg Niereand Matthias Tichy, Jörg Wadsack, Robert Wagnerand, Lothar Wendehals, , and Albert Zöndorf. Tool integration at the meta-model level: the fujaba approach. *Int. J. Softw. Tools Technol. Transf.*, 6(3):203–218, 2004.

[BHTV06] Luciano Baresi, Reiko Heckel, Sebastian Thöne, and Dániel Varró. Style-based modeling and refinement of service-oriented architectures. *Software and System Modeling*, 5(2):187–207, 2006.

[BKM08] Athman Bouguettaya, Ingolf Krüger, and Tiziana Margaria, editors. *Service-Oriented Computing - ICSOC 2008, 6th International Conference, Sydney, Australia, December 1-5, 2008. Proceedings*, volume 5364 of *Lecture Notes in Computer Science*, 2008.

[BLL$^+$95] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal - a tool suite for automatic verification of real-time systems. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer, 1995.

[BMSH10] Sebastian S. Bauer, Philip Mayer, Andreas Schroeder, and Rolf Hennicker. On Weak Modal Compatibility, Refinement, and the Mio Workbench. In *16th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010)*, 2010.

[Boo93] Grady Booch. *Object Oriented Analysis and Design with Applications.* Addison Wesley Professional, 10 1993.

[Buc05] Mikael Buchholtz. Automated analysis of infinite scenarios. In Rocco De Nicola and Davide Sangiorgi, editors, *TGC*, volume 3705 of *Lecture Notes in Computer Science*, pages 334–352. Springer, 2005.

[BV06] András Balogh and Dániel Varró. Advanced model transformation language constructs in the viatra2 framework. In *SAC*, pages 1280–1287, 2006.

[BZ07] Mario Bravetti and Gianluigi Zavattaro. A Theory for Strong Service Compliance. In Amy L. Murphy and Jan Vitek, editors, *9th Int. Conf. Coordination Models and Languages, COORDINATION 2007*, volume 4467 of *LNCS*, pages 96–112. Springer, 2007.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Snjiva Weerawarana. *Web Services Description Language (WSDL) 1.1.* World Wide Web Consortium, March 2001.

[CdAHS03] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Resource interfaces. In Rajeev Alur and Insup Lee, editors, *EMSOFT*, volume 2855 of *Lecture Notes in Computer Science*, pages 117–133. Springer, 2003.

[CGT09a] Allan Clark, Stephen Gilmore, and Mirco Tribastone. Quantitative analysis of web services using srmc. In Marco Bernardo, Luca Padovani, and Gianluigi Zavattaro, editors, *SFM*, volume 5569 of *Lecture Notes in Computer Science*, pages 296–339. Springer, 2009.

[CGT09b] Allan Clark, Stephen Gilmore, and Mirco Tribastone. Scalable analysis of scalable systems. In Marsha Chechik and Martin Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009.

[Cra04] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Prentice Hall, November 2004.

[dAH01a] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *Software Engineering Notes*, pages 109–120, 2001.

[dAH01b] Luca de Alfaro and Thomas A. Henzinger. Interface Theories for Component-Based Design. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *First Int. Workshop Embedded Software, EMSOFT 2001*, volume 2211 of *LNCS*, pages 148–165. Springer, 2001.

[dAH05] Luca de Alfaro and Thomas A. Henzinger. Interface-based Design. In Manfred Broy, Johannes Grünbauer, David Harel, and C. A. R. Hoare, editors, *Engineering Theories of Software-intensive Systems*, volume 195 of *NATO Science Series: Mathematics, Physics, and Chemistry*, pages 83–104. Springer, 2005.

[dAHS02] Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Timed interfaces. In Alberto L. Sangiovanni-Vincentelli and Joseph Sifakis, editors, *EMSOFT*, volume 2491 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2002.

[DBL88] *Proceedings, Third Annual Symposium on Logic in Computer Science, 5-8 July 1988, Edinburgh, Scotland, UK*. IEEE Computer Society, 1988.

[DFCU08] Nicolás D'Ippolito, Dario Fischbein, Marsha Chechik, and Sebastián Uchitel. MTSA: The Modal Transition System Analyser. In *23rd Int. Conf. Automated Software Engineering, ASE 2008*, pages 475–476. IEEE Computer Societey, 2008.

[DFFU07] Nicolás D'Ippolito, Dario Fischbein, Howard Foster, and Sebastián Uchitel. MTSA: Eclipse support for modal transition systems construction, analysis and elaboration. In Li-Te Cheng, Alessandro Orso, and Martin P. Robillard, editors, *OOPSLA Workshop Eclipse Technology eXchange, ETX 2007*, pages 6–10. ACM Press, 2007.

[DFK04] Jim D'Anjou, Scott Fairbrother, and Dan Kehn. *The Java Developers's Guide to Eclipse.* Addison-Wesley Longman, Amsterdam, November 2004.

[DGW08] Christophe Dumez, Jaafar Gaber, and Maxime Wack. Model-driven engineering of composite web services using uml-s. In Gabriele Kotsis, David Taniar, Eric Pardede, and Ismail Khalil Ibrahim, editors, *iiWAS*, pages 395–398. ACM, 2008.

[DHJP08] Laurent Doyen, Thomas A. Henzinger, Barbara Jobstmann, and Tatjana Petrov. Interface theories with component reuse. In Luca de Alfaro and Jens Palsberg, editors, *8th Int. Conf. Embedded software, EMSOFT 2008*, pages 79–88. ACM Press, 2008.

[DNsmGW08] Christophe Dumez, Ahmed Nait-sidi moh, Jaafar Gaber, and Maxime Wack. Modeling and specification of web services composition using uml-s. In *NWESP '08: Proceedings of the 2008 4th International Conference on Next Generation Web Services Practices*, pages 15–20, Washington, DC, USA, 2008. IEEE Computer Society.

[Ecl10a] Eclipse Foundation. EMF: The Eclipse Modeling Framework, 2010. http://eclipse.org/modeling/emf/.

[Ecl10b] Eclipse Foundation. GEF: The Graphical Editing Framework, 2010. http://www.eclipse.org/gef.

[Ecl10c] Eclipse Foundation. MDT: The Eclipse Model Development Tools Project, 2010. http://www.eclipse.org/modeling/mdt/.

[Ecl10d] Eclipse Foundation. MDT:UML2: The Eclipse Model Development Tools: UML2 Tools Subproject, 2010. http://www.eclipse.org/modeling/mdt/?project=uml2tools.

[Ecl10e] Eclipse Foundation. MoDisco: The Eclipse/GMT Model Discovery Component, 2010. http://www.eclipse.org/MoDisco/.

[Ecl10f] Eclipse Foundation. SWT: The Standard Widget Toolkit, 2010. http://www.eclipse.org/swt.

[Ecl10g] Eclipse Foundation. The Eclipse BPEL project, 2010. http://eclipse.org/bpel/.

[Ecl10h] Eclipse Foundation. The Eclipse Model To Text (M2T) Acceleo Subproject, 2010. http://www.eclipse.org/modeling/m2t/?project=acceleo.

[Ecl10i] Eclipse Foundation. The Eclipse Project, 2010. http://www.eclipse.org/.

[Ecl10j] Eclipse Foundation. WTP: The Eclipse Web Tools Platform Project, 2010. http://www.eclipse.org/webtools/.

[EGK$^+$10] Jannis Elgner, Stefania Gnesi, Nora Koch, , and Philip Mayer. *Specification and Implementation of Demonstrators for the Case Studies*, chapter 7.1. Springer Verlag, 2010.

[EK07] Vina Ermagan and Ingolf H. Krüger. A uml2 profile for service modeling. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 360–374. Springer, 2007.

[EN06] Greg Lomow Eric Newcomer. *Business Process Execution Language for Web Services.* Packt Publishing, 2006.

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall International, 2005.

[ES09] Jannis Elgner and Kamil Swierkot. Finance Case Study: Credit Portal. Technical report, s&n AG, 2009.

[Evi10] Eviware. SoapUI: Web Service Testing, 2010. http://www.soapui.org/.

[FBS04] Xian Fu, Tevfik Bultan, and Jianwen Su. Analysis of Interacting BPEL Web Services. In *3rd Int. Conf. on Web Services, ICWS 2004*, pages 621–630. IEEE Computer Society, 2004.

[FBU09] Dario Fischbein, Víctor A. Braberman, and Sebastián Uchitel. A Sound Observational Semantics for Modal Transition Systems. In Martin Leucker and Carroll Morgan, editors, *6th Int. Colloquium Theoretical Aspects of Computing, ICTAC 2009*, volume 5684 of *LNCS*, pages 215–230. Springer, 2009.

[FEK$^+$07] Howard Foster, Wolfgang Emmerich, Jeff Kramer, Jeff Magee, David Rosenblum, and Sebastian Uchitel. Model Checking Service Compositions under Resource Constraints. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the foundations of Software Engineering*, pages 225–234, New York, NY, USA, 2007. ACM.

[FGK$^+$10a] Howard Foster, László Göczy, Nora Koch, Philip Mayer, Carlo Montangero, and Dániel Varró. D1.4b: UML for Service-Oriented Systems. Specification, SENSORIA Project 016004, 2010.

[FGK$^+$10b] Howard Foster, László Gönczy, Nora Koch, Philip Mayer, Carlo Montangero, and Dániel Varró. *Sensoria: Software Engineering for Service-Oriented Overlay Computers*, chapter UML Extensions for Service-Oriented Systems. Springer Verlag, 2010.

[FGL$^+$08] Alessandro Fantechi, Stefania Gnesi, Alessandro Lapadula, Franco Mazzanti, Rosario Pugliese, and Francesco Tiezzi. A model checking approach for verifying COWS specifications. In J. L. Fiadeiro and P. Inverardi, editors, *11th Int. Conf. Fundamental Approaches to Software Engineering, FASE 2008*, volume 4961 of *LNCS*, pages 230–245. Springer, 2008.

[FM08] Howard Foster and Philip Mayer. Leveraging integrated tools for model-based analysis of service compositions. In Abdelhamid Mellouk, Jun Bi, Guadalupe Ortiz, Dickson K. W. Chiu, and Manuela Popescu, editors, *ICIW*, pages 72–77. IEEE Computer Society, 2008.

[FMRU08] Howard Foster, Arun Mukhija, David S. Rosenblum, and Sebastián Uchitel. A model-driven approach to dynamic and adaptive service brokering using modes. In Bouguettaya et al. [BKM08], pages 558–564.

[Fra03] David Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, January 2003.

[FUMK03] Howard Foster, Sebastián Uchitel, Jeff Magee, and Jeff Kramer. Model-based verification of web service compositions. In *ASE*, pages 152–163. IEEE Computer Society, 2003.

[FUMK06] Howard Foster, Sebastián Uchitel, Jeff Magee, and Jeff Kramer. Ltsa-ws: a tool for model-based verification of web service compositions and choreography. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 771–774. ACM, 2006.

[FUMK07] Howard Foster, Sebastián Uchitel, Jeff Magee, and Jeff Kramer. WS-Engineer: A Model-Based Approach to Engineering Web Service Compositions and Choreography. In Luciano Baresi and Elisabetta Di Nitto, editors, *Test and Analysis of Web Services*, pages 87–119. Springer, 2007.

[FUMK08] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Leveraging Modes and UML2 for Service Brokering Specifications. In *Proceedings of the 4th Model-Driven Web Engineering Workshop (MDWE 2008)*, Toulouse, France, 2008.

[FW04] David C. Fallside and Priscialla Walmsley. *XML Schema Part 0: Primer Second Edition.* World Wide Web Consortium, October 2004.

[GGK$^+$10] Stephen Gilmore, László Gönczy, Nora Koch, Philip Mayer, Mirco Tribastone, and Dániel Varró. Non-functional properties in the model-driven development of service-oriented systems. *Software and System Modeling*, 2010.

[GHM$^+$05] Olivier Gruber, B. J. Hargrave, Jeff McAffer, Pascal Rapicault, and Thomas Watson. The eclipse 3.0 platform: Adopting osgi technology. *IBM Systems Journal*, 44(2):289–300, 2005.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison Wesley, 2005.

[GL00] Stephen J. Garland and Nancy Lynch. Using I/O Automata for Developing Distributed Systems. In *In Gary T. Leavens and Murali Sitaraman, editors, Foundations of Component-Based Systems*, pages 285–312. Cambridge University Press, 2000.

[GMRMS09] Stephen Gorton, Carlo Montangero, Stephan Reiff-Marganiec, and Laura Semini. StPowla: SOA, Policies and Workflows. In Elisabetta Di Nitto and Matei Ripeanu, editors, *ICSOC'07 Workshops Revised Selected Papers*, volume 4907 of *Lecture Notes in Computer Science*, pages 351–362. Springer, 2009.

[GR01] Roberto Gorrieri and Arend Rensink. Action refinement. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, chapter 16, pages 1047–1147. Elsevier, 2001.

[GR03] Anna Gerber and Kerry Raymond. Mof to emf: there and back again. In Michael G. Burke, editor, *OOPSLA Workshop on Eclipse Technology eXchange*, pages 60–64. ACM, 2003.

[GV10] László Gönczy and Dániel Varró. *Developing Effective Service Oriented Architectures: Concepts and Applications in Service Level Agreements, Quality of Service and Reliability*, chapter Engineering Service Oriented Applications with Reliability and Security Requirements. IGI Global, 2010. To be published.

[HÖ7] Matthias Hölzl. D8.4a: Distributed E-University Management and E-Learning System: Requirements modelling and analysis of selected scenarios. Deliverable for the eu project sensoria, reporting period october 2006 - september 2007, SENSORIA Project 016004, 2007.

[Hil96] Jane Hillston. A compositional approach to performance modelling, 1996.

[HKMU06] Dan Hirsch, Jeff Kramer, Jeff Magee, and Sebastian Uchitel. Modes for Software Architectures. In *Proceedings of EWSA 2006, 3rd European Workshop on Software Architecture*, Lecture Notes in Computer Science. Springer Verlag, 2006.

[HL89] Hans Hüttel and Kim Guldstrand Larsen. The Use of Static Constructs in A Modal Process Logic. In Albert R. Meyer and Michael A. Taitslin, editors, *Symp. Logical Foundations of Computer Science, Logic at Botik 1989*, volume 363 of *LNCS*, pages 163–180. Springer, 1989.

[HLT03] Reiko Heckel, Marc Lohmann, and Sebastian Thne. Towards a uml profile for service-oriented architectures. In *Workshop on Model Driven Architecture: Foundations and Applications*, 2003.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual.* Addison-Wesley Professional, 2003.

[IBM09] IBM Corporation. Rational Software Architect for WebSphere Software, 2009. http://www-01.ibm.com/software/awdtools/swarchitect/websphere/.

[Jac92] Ivar Jacobson. *Object Oriented Software Engineering: A Use Case Driven Approach.* Bennet Books Ltd, 07 1992.

[Jam02] James Clark. RELAX NG and W3C XML Schema, 2002. http://www.imc.org/ietf-xml-use/mail-archive/msg00217.html.

[JBo10] JBoss Community. JBoss jBPM - Workflow in Java, 2010. http://www.jboss.org/jbpm.

[JMS03] Matjaz Juric, Beny Mathew, and Poornachandra Sarang. *Web Services: Concepts, Architectures, and Applications.* Springer, Berlin, 2003.

[Joh05] Simon Johnston. UML 2.0 Profile for Software Services. Technical report, IBM Corporation, Apr 2005. `https://www.ibm.com/developerworks/rational/library/05/419_soa/`.

[KMH$^+$07] Nora Koch, Philip Mayer, Reiko Heckel, László Göczy, and Carlo Montangero. D1.4a: UML for Service-Oriented Systems. Specification, SENSORIA Project 016004, 2007.

[Lin06] Fabian Linz. Integration von Benutzeraktionen in BPEL-Prozesse. Masters thesis, Universitt Paderborn (in Kooperation mit der S&N AG), 2006.

[LMS08] Nancy A. Lynch, Laurent Michel, and Alexander Shvartsman. Tempo: A Toolkit for The Timed Input/Output Automata Formalism. In Sándor Molnár, John Heath, Olivier Dalle, and Gabriel A. Wainer, editors, *1st Int. Conf. Simulation Tools and Techniques for Communications, Networks, and Systems, SimuTools 2008.* ICST, 2008.

[LMSW06] Niels Lohmann, Peter Massuthe, Christian Stahl, and Daniela Weinberg. Analyzing interacting BPEL processes. In Schahram Dustdar, José Luiz Fiadeiro, and Amit P. Sheth, editors, *4th Int. Conf. Business Process Management, BPM 2006*, volume 4102 of *LNCS*, pages 17–32. Springer, 2006.

[LNW07a] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal I/O Automata for Interface and Product Line Theories. In Rocco De Nicola, editor, *16th Eur. Symp. Programming, Programming Languages and Systems, ESOP 2007*, volume 4421 of *LNCS*, pages 64–79. Springer, 2007.

[LNW07b] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. On Modal Refinement and Consistency. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *18th Int. Conf. Concurrency Theory, CONCUR 2007*, volume 4703 of *LNCS*, pages 105–119. Springer, 2007.

[LPT09] Alessandro Lapadulaa, Rosario Pugliese, and Francesco Tiezzi. Using formal methods to develop WS-BPEL applications. Technical report, Dipartimento di Sistemi e Informatica, University of Firenze, 2009.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *6th Annual Symp. Principles of Distributed Computing, PODC 1987*, pages 137–151. ACM Press, 1987.

[LT88a] Kim Guldstrand Larsen and Bent Thomsen. A Modal Process Logic. In *3rd Annual Symp. Logic in Computer Science, LICS 1988* [DBL88], pages 203–210.

[LT88b] Kim Guldstrand Larsen and Bent Thomsen. A modal process logic. In *LICS* [DBL88], pages 203–210.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[LZP09] Bixin Li, Yu Zhou, and Jun Pang. Model-driven automatic generation of verified bpel code for web service composition. In Shahida Sulaiman and Noor Maizura Mohamad Noor, editors, *APSEC*, pages 355–362. IEEE Computer Society, 2009.

[Man03] Keith Mantell. From UML to BPEL: Model Driven Architecture in a Web Services world. Technical report, International Business Machines Corporation (IBM), September 2003.

[Mar05] Axel Martens. Analyzing web service based business processes. In Maura Cerioli, editor, *8th Int. Conf. on Fundamental Approaches to Software Engineering, FASE 2005*, volume 3442 of *LNCS*, pages 19–33. Springer, 2005.

[MB07] Philip Mayer and Hubert Baumeister. D7.4b: Report on the Sensoria CASE Tool: Description and Evaluation. Deliverable for the eu project sensoria, reporting period october 2006 - september 2007, SENSORIA Project 016004, 2007.

[MCF03] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Guest editors' introduction: Model-driven development. *IEEE Software*, 20(5):14–18, 2003.

[MCG05] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. A taxonomy of model transformations. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proc. ESEC 1995*, volume 989 of *LNCS*, pages 137–153. Springer, 1995.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall (International Series in Computer Science), 1989.

[MK99] Jeff Magee and Jeff Kramer. *Concurrency: state models & Java programs.* John Wiley & Sons, Inc., New York, NY, USA, 1999.

[MNS05] Tiziana Margaria, Ralf Nagel, and Bernhard Steffen. Remote integration and coordination of verification tools in jeti. In *ECBS*, pages 431–436. IEEE Computer Society, 2005.

[MR09] Jim Marino and Michael Rowley. *Understanding SCA (Service Component Architecture).* Addison-Wesley Longman, July 2009.

[MR10] Philip Mayer and Istvan Rath. D7.4d: Report on the Sensoria Development Environment (SDE), third version. Deliverable for the eu project sensoria, reporting period october 2008 - february 2010, SENSORIA Project 016004, 2010.

[MRH08] Philip Mayer, Istvan Rath, and Adam Horvath. D7.4c: Report on the Sensoria Development Environment (SDE), second version. Deliverable for the eu project sensoria, reporting period october 2007 - september 2008, SENSORIA Project 016004, 2008.

[MS08] Tiziana Margaria and Bernhard Steffen, editors. *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15, 2008. Proceedings*, volume 17 of *Communications in Computer and Information Science*. Springer, 2008.

[MSB10] Philip Mayer, Andreas Schroeder, and Sebastian S. Bauer. A Strict-Observational Interface Theory for Analysing Service Orchestrations. In *7th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2010)*, 2010.

[MSK08a] Philip Mayer, Andreas Schroeder, and Nora Koch. Mdd4soa: Model-driven service orchestration. In *EDOC*, pages 203–212. IEEE Computer Society, 2008.

[MSK08b] Philip Mayer, Andreas Schroeder, and Nora Koch. A model-driven approach to service orchestration. In *IEEE SCC (2)*, pages 533–536. IEEE Computer Society, 2008.

[NoM10] NoMagic, Inc. MagicDraw, 2010. http://www.magicdraw.com/.

[NW04] Steve Northover and Mile Wilson. *SWT: The Standard Widget Toolkit, Volume 1.* Addison-Wesley Professional, July 2004.

[OAS06] OASIS. *Reference Model for Service-Oriented Architecture 1.0.* Organization for the Advancement of Structured Information Standards, Billerica, USA, August 2006.

[OAS07] OASIS. *Web Services Business Process Execution Language Version 2.0.* Organization for the Advancement of Structured Information Standards, Billerica, USA, April 2007.

[OAS10] OASIS. Organization for the Advancement of Structured Information Standards. Technical report, OASIS, 2010. http://www.oasis-open.org/.

[OMG06] OMG (Object Management Group). UML Profile and Metamodel for Services (UPMS) Request For Proposal, 09 2006.

[OMG07] OMG (Object Management Group). MOF 2.0/XMI Mapping, Version 2.1.1. Specification, OMG (Object Management Group), 12 2007. `http://www.omg.org/cgi-bin/doc?formal/07-12-01.pdf`.

[OMG08a] OMG (Object Management Group). A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 2. Technical report, OMG (Object Management Group), 2008.

[OMG08b] OMG (Object Management Group). *MOF Model to Text Transformation Language (MOFM2T), 1.0.* OMG (Object Management Group), Needham, USA, January 2008.

[OMG09a] OMG (Object Management Group). Business Process Model and Notatation (BPMN). Specification, OMG (Object Management Group), 1 2009. `http://www.omg.org/spec/BPMN/1.2/`.

[OMG09b] OMG (Object Management Group). Service oriented architecture Modeling Language(SoaML), Beta 2. Technical report, OMG (Object Management Group), 2009.

[OMG10a] OMG (Object Management Group). Unified Modeling Language: Infrastructure, version 2.3. Technical report, OMG (Object Management Group), 2010.

[OMG10b] OMG (Object Management Group). Unified Modeling Language Superstructure. Specification, OMG (Object Management Group), 5 2010. `http://www.omg.org/spec/UML/2.3/Superstructure/`.

[Ope04] Open Source Initiative (OSI)). Eclipse Public License -v 1.0. License, OSI, May 2004. `http://www.opensource.org/licenses/eclipse-1.0.php`.

[Ora10] Oracle. Oracle BPEL Process Manager, 2010. http://www.oracle.com/technology/products/ias/bpel/.

[OSG08] OSGi Alliance. Osgi specification release 4. `http://www.osgi.org/Specifications/`, 03 2008.

[OSI10] OSI. OSI: The Open Source Initiative, 2010. http://www.opensource.org/.

[Par07] Terence Parr. *The Definitivie ANTLR Reference Guide: Building Domain-Specific Languages.* Pragmatic Programmers, May 2007.

[PRO05] PROFUNDIS project team. PROFUNDIS: Proofs of Functionality for Mobile Distributed Systems, 2005. http://www.it.uu.se/profundis/.

[RAR07] Jan S. Rellermeyer, Gustavo Alonso, and Timothy Roscoe. R-OSGi: distributed applications through software modularization. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 1–20, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[RBB⁺09a] Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, Axel Legay, and Roberto Passerone. Modal interfaces: unifying interface automata and modal specifications. In Samarjit Chakraborty and Nicolas Halbwachs, editors, *9th Int. Conf. Embedded software, EMSOFT 2009*, pages 87–96. ACM Press, 2009.

[RBB⁺09b] Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benot Caillaud, and Roberto Passerone. Why Are Modalities Good for Interface Theories? In *9th Int. Conf. Application of Concurrency to System Design, ACSD 2009*, pages 119–127, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[RBL⁺90] James Rumbaugh, Michael Blaha, William Lorensen, Frederick Eddy, and William Premerlani. *Object-Oriented Modeling and Design.* Bennet Books Ltd, 10 1990.

[Ric00] Richard Soley. Model Driven Architecture. Technical report, OMG (Object Management Group), 2000.

[RM06] Jan Recker and Jan Mendling. On the translation between bpmn and bpel: Conceptual mismatch between process modeling languages, 2006.

[SD05] Zoran Stojanovic and Ajantha Dahanayake. *Service Oriented Software System Engineering: Challenges and Practices.* Idea Group Publishing, April 2005.

[SeC10] SeCSE project team. Service Centric System Engineering (SeCSE), 2010. http://www.secse-project.eu/.

[Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[SGS04] David Skogan, Roy Grønmo, and Ida Solheim. Web service composition in uml. In *EDOC*, pages 47–57. IEEE Computer Society, 2004.

[SKM01] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking uml state machines and collaborations. *Electr. Notes Theor. Comput. Sci.*, 55(3), 2001.

[SM08] Andreas Schroeder and Philip Mayer. Verifying interaction protocol compliance of service orchestrations. In Bouguettaya et al. [BKM08], pages 545–550.

[soa07] *SOAP* . World Wide Web Consortium (W3C), April 2007.

[tBBG07] Maurice H. ter Beek, Antonio Bucchiarone, and Stefania Gnesi. Web Service Composition Approaches: From Industrial Standards to Formal Methods. In *2nd Int. Conf. Internet and Web Applications and Services, ICIW 2007*. IEEE Computer Society, 2007.

[tBMG09] Maurice H. ter Beek, Franco Mazzanti, and Stefania Gnesi. Cmcumc: a framework for the verification of abstract service-oriented properties. In Sung Y. Shin and Sascha Ossowski, editors, *SAC*, pages 2111–2117. ACM, 2009.

[TG08] Mirco Tribastone and Stephen Gilmore. Automatic translation of uml sequence diagrams into pepa models. In *QEST*, pages 205–214. IEEE Computer Society, 2008.

[TG10] Mirco Tribastone and Stephen Gimore. *Sensoria: Software Engineering for Service-Oriented Overlay Computers*, chapter Scaling Performance Analysis using Fluid-Flow Approximation. Springer Verlag, 2010.

[W3C10] W3C. World Wide Web Consortium. Technical report, W3C, 2010. http://www.w3.org/.

[WBC$^+$09] Martin Wirsing, Laura Bocchi, Alan Clark, Jose Fiadeiro, Stephen Gilmore, Matthias Hölzl, Nora Koch, Philip Mayer, Roberto Pugliese, and Andreas Schroeder. *At your service: Service Engineering in the Information Society Technologies Program*, chapter Sensoria: Engineering for Service-Oriented Overlay Computers. MIT Press, 2009.

[WCL$^+$05] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More.* Prentice Hall, Upper Saddle River, New Jersey, USA, 2005.

[WHA$^+$08] Martin Wirsing, Matthias M. Hölzl, Lucia Acciai, Federico Banti, Allan Clark, Alessandro Fantechi, Stephen Gilmore, Stefania Gnesi, László Gönczy, Nora Koch, Alessandro Lapadula, Philip Mayer, Franco Mazzanti, Rosario Pugliese, Andreas Schroeder, Francesco Tiezzi, Mirco Tribastone, and Dániel Varró. Sensoriapatterns: Augmenting service engineering with formal analysis, transformation and dynamicity. In Margaria and Steffen [MS08], pages 170–190.

[WHK$^+$08] Martin Wirsing, Matthias Hölzl, Nora Koch, Philip Mayer, and Andreas Schroeder. Service engineering: The sensoria model driven approach. In *Proceedings of Software Engineernig Research, Managaement and Applications (SERA)*, Prague, Czech Republic, August 2008.

[Wor04a] World Wide Web Consortium (W3C). Web Services Architecture. Technical report, World Wide Web Consortium (W3C), 2004.

[Wor04b] World Wide Web Consortium (W3C). Web Services Architecture Requirements. Technical report, World Wide Web Consortium (W3C), 2004.

[wsa04] *Web Services Addressing (WS-Addressing)* . World Wide Web Consortium (W3C), August 2004.

[XK09] Rong Xie and Nora Koch. Automotive Case Study: Demonstrator. Technical report, Cirquent GmbH, 2009.

List of Tables

List of Figures

List of Algorithms

List of Listings

List of Definitions