Toni Reichelt

A Model Driven Approach for Service Based System Design Using Interaction Templates

Wissenschaftliche Schriftenreihe

EINGEBETTETE, SELBSTORGANISIERENDE SYSTEME

Band 10

Prof. Dr. Wolfram Hardt (Hrsg.)

Toni Reichelt

A Model Driven Approach for Service Based System Design Using Interaction Templates

CHEMNITZ UNIVERSITY OF TECHNOLOGY

Chemnitz University Press
2012

Impressum

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Angaben sind im Internet über http://dnb.d-nb.de abrufbar.

Zugl.: Chemnitz, Techn. Univ., Diss., 2012

Technische Universität Chemnitz/Universitätsbibliothek
Universitätsverlag Chemnitz
09107 Chemnitz
http://www.bibliothek.tu-chemnitz.de/UniVerlag/

Herstellung und Auslieferung
Verlagshaus Monsenstein und Vannerdat OHG
Am Hawerkamp 31
48155 Münster
http://www.mv-verlag.de

ISBN 978-3-941003-60-6

URL: http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-85986

Preface to the scientific series “Eingebettete, selbstorganisierende Systeme”

The tenth volume of the scientific series *Eingebettete, selbstorganisierende Systeme* (Embedded, self-organised systems) devotes itself to the model driven design of service-oriented systems.

Fields of application for such systems, amongst others, range from large scale, distributed enterprise solutions down to modularised embedded systems. The presented work was motivated by the complex design process of avionics for unmanned systems, where the increased demand for enhanced system autonomy implies high levels of system collaboration and information exchange. For such system designs a clear separation between the application logic implemented by system components, the means of interaction between them and finally the realisation of such communication on target platforms is essential. In particular, this separation leads to simplification of the complex design processes and increases the individual components’ re-use potential.

With the presented work, Dr. Reichelt provides a holistic model driven design process for such systems. The design of service oriented systems and their target platform independent modelling with respect to component interaction stay in the focus of his work. The concrete semantics of individual interactions are condensed into interaction templates collected in an extensible model library. In turn, this library then provides the building blocks for service and system modelling by means of template instantiation. Dr. Reichelt reflects his formal concepts in a comprehensive UML profile and presents an encapsulating design process. This new design process is validated with a representative case study based on a prototyped tooling environment comprising model transformation and code generation techniques.

I am glad that Dr. Reichelt publishes his research in this scientific series and wish you an interesting insight to the field of model driven design of service-oriented systems.

[illegible]

Prof. Dr. Wolfram Hardt
Chair of Computer Engineering
Dean of the Faculty of Computer Science
Scientific Director of Computing Center
Chemnitz University of Technology, Germany
April 2012

CHEMNITZ UNIVERSITY OF TECHNOLOGY

A Model Driven Approach for Service Based System Design Using Interaction Templates

Dissertation

Submitted in partial fulfilment of
the requirements for the degree of

Doktoringenieur
(Dr.-Ing.)

presented to the

Faculty of Computer Science
Chemnitz University of Technology

Submitted by
Toni Reichelt

Assessor: Prof. Dr. Wolfram Hardt
Prof. Dr.-Ing. Martin Gaedke

Acknowledgements

This thesis would not exist without the extensive support and encouragement by many people. First of all I am grateful to my mentor Prof. Dr. Wolfram Hardt who supported me in pursuing a PhD degree. Within fruitful discussions he helped shaping and organising many of the ideas of this thesis. And as this thesis was prepared in an industrial environment I am especially thankful because he always welcomed me in academia and allowed me to stay in touch with the scientific world.

I would also like to thank Prof. Dr.-Ing. Martin Gaedke for his interest in this thesis and very constructive feedback in structuring and improving the presented work.

It is an honour for me to thank some personal friends with whom I had the pleasure to work with at Cassidian. In particular I am indebted to Dr. André Windisch and Dr. Norbert Oswald. André, thank you for introducing me to Cassidian and the world of avionics. Norbert, I am thankful for given me the opportunity to join the *EPAS* (*Engineering Platform for Autonomous Systems*) team which provided an ideal framework for developing my ideas and test driving solution concepts.

Also, I thank my friends Dr.-Ing. Stefan Förster and Herwig Moser for their almost endless willingness to discuss many ideas compiled into this work as well as proof-reading and commenting on its final version.

Finally, I would also like to thank my family, especially my wife Conny. Without their continous support this work would not have been possible.

Thank you.

Toni Reichelt
Munich, August, 2011

Abstract

Based on the increasing complexity of modern avionics, the associated system design processes moved towards *Model Driven Architecture* (*MDA*) based processes. Additionally, the demand for higher system autonomy features requires means to further modularise mission systems and to define and establish interactions among the systems' individual components. Therefore, the ideas of service-oriented computing are currently adapted to established, model driven design processes. With respect to modelling interactions for service components, current approaches are limited to only a fixed set of communication primitives, restricting a service designer's expressiveness to specify service interaction. In consequence, interaction patterns not included in this basic set have to be reflected in application code, mangling application and communication logic. Furthermore, when service functionality relies on communication semantics which are not provided by the underlying set of primitives, additional emulation behaviour has to be added to the service which makes this mangling even worse. Platform independence is reduced as services can not easily be ported to platforms not natively supporting the selected primitives which contradicts the ideas of model-driven development.

Addressing these limitations, this thesis proposes a new model-driven service development process based on *Interaction Templates* (*ITs*), promoting interactions among service participants to first class modelling entities. The process focuses on modelling the interactions among service participants. Interaction semantics are explicitly specified in models, beyond pure stereotyping, and gain increased platform independence for services with respect to communication. The process exploits automated *Model-to-Model* (*M2M*) and *Model-to-Text* (*M2T*) transformations to assist service implementation and to automatically derive interaction realisations on concrete target platforms. This allows for easy replacement and inter-mixing of communication middleware to realise a service's interactions. This way, services become independent of the underlying communication primitives by only relying on ITs and not platform primitives which are hidden behind ITs. In turn, realising ITs on concrete platforms is not affected by their utilisation for service interaction.

Beside the novel modelling process itself, the presented work defines a *Unified Modeling Language* (*UML*) profile, referred to as *UML Profile for Interaction-centric Services* (*UP4IS*), which directly supports the adaptation of standard UML language constructs and tools for the proposed modelling approach. The whole development process is demonstrated via the specification of a simple video recording systems consisting of two services. The services themselves are based on a representative IT library which forms an essential part of the presented case study. Using these service and IT models, the thesis emphasises the necessary model transformation and code generation steps to derive service implementations based on the abstract models.

Contents

List of Tables

List of Figures

List of Listings

List of Acronyms

ADL	Architecture Description Language.
API	Application Programming Interface.
ATL	Atlas Transformation Language.
BPEL	Business Process Execution Language.
BPEL4WS	BPEL for Web Services.
BPMN	Business Process Modelling Notation.
CIM	Computation Independent Model.
CORBA	Common Object Request Broker Architecture.
DSL	Domain Specific Language.
ESB	Enterprise Service Bus.
FSM	Finite State Machine.
IDL	Interface Description Language.
ISO	International Standards Organisation.
IT	Interaction Template.
ITU	International Telecommunication Union.
JAR	Java Archive.
JAUS	Joint Architecture for Unmanned Systems.
JP2K	JPEG 2000.
JPEG	Joint Photographic Experts Group.
M2M	Model-to-Model.
M2T	Model-to-Text.
MDA	Model Driven Architecture.
MOF	Meta Object Facility.

MOF-M2T	MOF Models to Text Transformation Language.
MSC	Message Sequence Chart.
OASIS	Organization for the Advancement of Structured Information Standards.
OCL	Object Constraint Language.
OMG	Object Management Group.
OMT	Object Modelling Technique.
OOD	Object-Oriented Design.
OOSE	Object-Oriented Software Engineering.
OSGi	Open Services Gateway Initiative.
OSI	Open Systems Interconnection.
PIM	Platform Independent Model.
PSM	Platform Specific Model.
QoS	Quality-of-Service.
QVT	Query/View/Transformations.
R/R	Request/Response.
RMI	Remote Method Invocation.
RPC	Remote Procedure Call.
SOA	Service Oriented Architecture.
SSD	System Sequence Diagram.
UML	Unified Modeling Language.
UML4SOA	UML Profile for SOA.
UP4IS	UML Profile for Interaction-centric Services.
URML	UML based Rule Modelling Language.
WSDL	Web Service Description Language.
WSTL	Web Service Transition Language.
XMI	XML Metadata Interchange Format.

CHAPTER 1

Introduction

1.1 Motivation and Application Context

Driven by the findings and achievements in the domains of artificial intelligence and robotics in the last decade, the development and application of unmanned systems become a popular alternative for mission scenarios which are considered as too risky for execution by humans, referred to as “dull, dirty and dangerous missions” [FHS04], e.g., damage assessment after natural or nuclear disasters. Especially the aeronautics industry is challenged by the increasing demands on system autonomy which implies a high level of system collaboration and information exchange within as well as in-between the participating systems. A key element for such systems is not only to realise some functionality but also the ability to provide resources and capabilities to other system [Alb03]. This requirement leads to more and more complex avionics system designs. In turn, system designers are now faced with the trade-off between the traditional avionics design process, driven by safety and security regulations [Spi00], and the requirements to design modular and extensible systems [WFF02].

To face these challenges, the development processes are adjusted to incorporate established standards from domains outside avionics, most prominent software and distributed systems design. Mainly, the processes were aligned on model driven approaches [Spi06] as represented by the MDA [OMG01], e.g., in form of frameworks like *Joint Architecture for Unmanned Systems* (*JAUS*) [JAU06] and modelling tools[1]. The adaption of MDA yielded a design process applicable to the complete system life cycle, from the abstract specification, through the implementation, up to maintenance and re-design [MM03].

However, current process adaptations focus more on architectural facets of the development process [Spi06]. What is blended out is the description of a system’s resources and capabilities and how these are accessed during mission execution. It was shown, that service oriented design can be applied to avionics, especially for autonomous system design [OWF$^+$07, OFM$^+$07]. Not only provides it a way to address the before mentioned modelling problem, but also offers the possibility to incorporate even more, complementing

[1]For instance IBM Rational Rhapsody, see `http://www-01.ibm.com/software/awdtools/rhapsody/`.

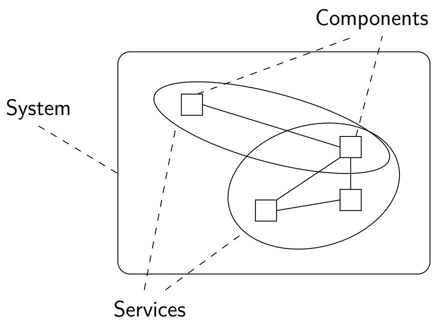


Figure 1.1: The relationship between systems, services, and components.

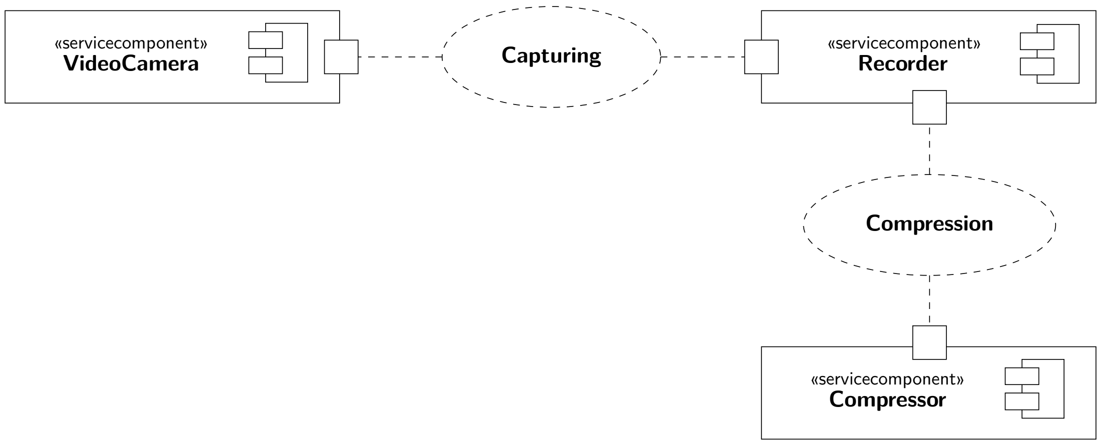


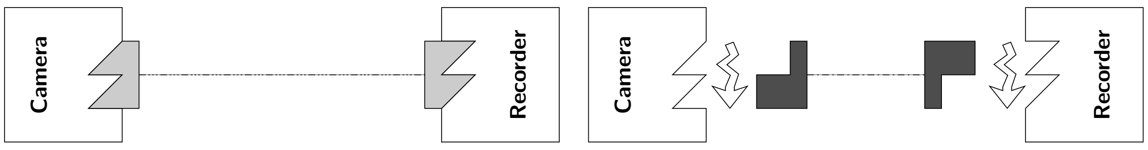
Figure 1.2: A simple video recorder.

technologies for avionics, e.g., information dissemination [MROF08, MORF09] or automated planning and plan execution [MROF09a, MROF09b].

1.2 Problem Description

A service represents a distinct unit of system logic resulting from the decomposition of a system's behaviour to separate functional groups. Services are implemented by components which can be individually distributed. A service oriented system then evolves from loosely coupled compositions of such components [MLM^{+}06] (cf. Figure 1.1). "Collectively, these [components] comprise a larger piece of business automation logic. Individually, these [components] can be distributed." [Erl05, p. 32] Hence, an essential part of service orientation is to establish communication between the distributed components. To draw emphasis on this aspect of service orientation, the notion of services is understood as *interaction-centric service* within the presented thesis, i.e., it represents the collaboration of entities – it describes collaborative functionality [DMRK05, BF05, KM04].

Figure 1.2 depicts a simple video recorder system which may be realised for an unmanned system to acquire image data for later analysis. The video recorder is composed out of two services, capturing and compression. The system shall capture image data through a video camera. The data shall be stored by a recorder component in a compressed manner. This compression functionality is provided by a third component, the compressor.



(a) Middleware specific service interactions. (b) Inability to replace middleware.

Figure 1.3: Platform dependent modelling of services.

Once, the service components are modelled, implemented and deployed to a system, system behaviour is only realised if the necessary communication between the various components can be established at run-time. Hereby, one often focuses on specific interaction features of the addressed deployment platform a system is initially designed for [Spi06]. Often, possible invocation mechanisms to be exploited for service development are restricted to a "widely accepted standard" [Pap03] and thus to a limited set of interaction primitives. Typically, this set of primitives is referred to as abstract platform [Alm06]. Although concrete technologies which can be used to realise these interactions are generalised, the abstract platform still remains quite specific as it constraints communication to small sets of applicable primitives. This contradicts the already established principles of MDA for avionics as the system designer has to handle details of the target platform, e.g., message passing mechanisms and data representation, within the high level system specifications which is intentionally to be avoided within MDA [VÖ7, p. 424].

Figure 1.3 illustrates the consequences of such platform dependent modelling. In Figure 1.3a. the initially designed components of the camera and recorder are shown. Both are interconnected by some middleware which realises communication between them, e.g., to stream the video data. The component's interfaces are based on the characteristics of that middleware, exemplified by the jagged shape of the middleware adaptors, shown in light grey. If the middleware shall be replaced, e.g., to adapt the components to a new system, and the new middleware does not provide the same interaction semantics as illustrated by a different adaptor shape in Figure 1.3b, the components could not be re-used without modification of their own interfaces. The reason for this is, that platform dependent information flew implicitly into the component models which was not reflected as first class modelling entity [KBW03, p. 2]. This is an evident problem when regarding system interoperability and portability ([KBW03, p. 5] and [ROW$^+$07]).

What would be beneficial is a modelling process to be integrated into existing MDA processes which would not only allow for explicit modelling of a system's services but also to support direct modelling of service interactions with respect to the underlying interaction's semantics of in a platform independent manner. This will increase re-usability of service components due to increased platform independence. Furthermore, this approach supports grounding of service interactions to different platforms (middleware) solutions based on the semantics of the individual service interactions rather that being limited by a single platform's support for very specific semantics. The MDA process would immediately enable the integration of different target platforms not only as mutually exclusive mapping alternatives but more prominent as specialised options to be used in parallel in a heterogeneous system execution environment. This way, the individual strengths of different middleware can be combined to form a communication environment specifically tailored to the modelled system[2]. Such tailoring on interaction level rather than on service level is especially beneficial in embedded environments, such as avionics, which challenge system designers with various limitations like restricted communication stacks and middleware solutions (cf. [VCH10, VE10]). Additionally, a modelling approach clearly separating between a service's functional design and the service's interaction semantics will ease further extensions of the development process, e.g., adding support for resource oriented methodologies beside service orientation [HGM09] or, of special interest for safety and security critical systems, authorisation and access control enforcement beyond system boundaries [HGM08, Mei09] to be plugged into interaction models.

1.3 Solution Requirements and Contributions

Within the context of MDA based service development, the presented thesis describes a model driven development process based on service interaction patterns represented as *first class modelling entities*, referred to as *Interaction Templates* (*ITs*). By using ITs as building blocks for service specification, the restriction to a closed set of interaction primitives is eliminated and real platform independence with respect to communication realisation in accordance with MDA is gained. Therefore, the proposed approach diminishes the influences of platform restrictions on service design. Based on ITs, a service's interactions are explicitly designed for a service's needs rather than being governed by a platform's characteristics. The concept of IT oriented service development is embedded in a comprehensive design methodology enclosing a complete MDA process chain which supports code generation for service component implementation and automated groundings of service interactions to target platforms. Hence, the described process simplifies a service's specification by maintaining a strict separation of service functionality, service interaction and the service deployment platform. In particular, this separation yields the following set of advantages:

- Primitive as well as complex interactions are separated from a service's application/business logic and the concrete features of target platforms.
- Service modelling is unconstrained by the underlying platform's communication primitives which remain hidden by ITs.
- Interactions are automatically realised on target platforms based on their usage within a service specification.
- Interactions can be mapped to multiple target platforms, allowing for simplified replacement without affecting existing service implementations.
- Individual interactions of one service can be realised through different target platforms, exploiting specialised communication primitives where possible.

In order to achieve these benefits, the modelling methodology described within this thesis contributes answers to the following requirements on an MDA-based interaction-centric service development process:

[2]Cf. [Lin06, Ö06] for a more detailed discussion on combining standard middleware for the specific needs of a system.

- A *pattern language* for service interaction to support the creation of an IT library to base service development on.

  Supporting an open approach to define and extend the IT library as core element of the process, does not restrict the process to specific fields of application.

- A *formal model*
  - for the *structural and behavioural specification of service interaction patterns* between a pair of communicating entities,
  - *allowing for service definition* with two or more participants based on interaction patterns,
  - *supporting component and system specification* based on services.

  Supporting a formal representation of the modelled entities allows support for system validation and verification which is crucial for extending avionics development processes.

- An *UML profile* reflecting this formal model and thus supporting interaction (pattern), service, and system design with standard UML tools.

  Integration of the formal concepts into UML enables application of the process to already established UML/MDA based development processes and tools and thus will ease the adaptation of new processes.

- A *generic concept for automatic realisation of service interactions* on target platforms gaining platform independence with respect to communication between service participants and to support automated grounding of service interactions.

  By not being bound to specific target platforms the process supports increased re-use of the modelled services and interactions. By also enabling the automated derivation of implementations of service interactions, the process will simplify the overall development process.

- An *enclosing MDA process* for service-oriented system development integrating the above approaches with automated model transformation and code generation.

The applicability of our approach is demonstrated by a representative case study which comprises the creation of a comprehensive pattern library for service interaction and its usage for the specification of the already mentioned video recording system based on distinct services. Additionally, the relevant model transformations and parts of the generated source code are presented.

1.4 Outline

The thesis is structured as follows:

Part I – Foundations. Chapter 2 presents the discussion of related work with respect to the contributions of this thesis. Relevant literature in the field of service specification, formal models for service orientation in general and for service interaction in particular, as well as work exploiting UML in the context of service modelling is highlighted. The remaining chapters of Part I present selected concepts of MDA and UML which are relevant for presented development process.

Part II – Modelling Interaction-Centric Services. Part II describes the methodology of interaction-centric service development based on ITs. In Chapter 5 the formal modelling framework driving the design process is presented along with the visualisation approach for the introduced formal concepts based on UML, given by the UML profile specified in Chapter 6.

Part III – Case Study. The process' modelling approach is demonstrated by developing a basic video recording system based on our design methodology. Therefore, a representative set of interaction patterns in form of an IT library in Chapter 7 is developed. This library is then exploited in Chapter 8 to define the exemplified services and the video recording system itself. Concluding the case study, the necessary model transformations to support the implementation of the services on the one hand as well as to enable the automatic realisation of service interaction on target platforms on the other hand are explained. (Chapter 9).

Conclusions. Eventually, Chapter 10 concludes by summarising the main contributions of the thesis and outlining open challenges for future research.

Part I

Foundations

CHAPTER 2

Related Work

This chapter categorises and discusses related work in the field of service oriented system design and the description of service interactions. Thereby, the discussion focuses on the requirements identified in Section 1.3, especially highlighting work associated to the MDA/UML context.

2.1 Modelling Service Oriented Systems

Modelling services represents a vast field of system design. Therefore, the presented approaches are representatives for whole categories, each emphasising a different aspect of service modelling. In particular, the following sections introduce formal frameworks for service (component) specification, reference models addressing complete service architectures, and finally work in relation to model driven service development based on UML.

2.1.1 Formal Frameworks

In literature exist several competing formal approaches for service-oriented system development. The formalisation is largely driven by the need for a well defined semantics in order to enable machine processibility. Hence, one can exploit automated validation checks for system definitions, e.g., proofing their soundness, timing constraints, or coverage of component requirements. Formal frameworks for service-oriented system development can be categorised mainly in two different manners: First, if they are founded on common, well known formalisms like process algebras or automata theory, or if they contribute their own formal languages and semantics. Second, they can be distinguished by addressing complete architectures, i.e., combining static and dynamic aspects, or by focusing primarily on behavioural specifications.

Derived Formalisms

In the field of service specification mainly the following formalisms are used for behaviour description: *Finite State Machines* (*FSMs*) [Gil62], Petri-Nets [Pet81, Jen90], and *Mes-*

sage Sequence Charts (*MSCs*) [oI01]. A typical representative of the former is the *Web Service Transition Language* (*WSTL*) proposed by Berardi et al. in [BdRdSM04]. WSTL uses finite state automata to describe "what is observable from the point of view of the service users" [BdRdSM04]. It describes service components based on communication scenarios, referred to as *conversations*, and the resulting transitions in the surrounding system. WSTL is a derivation of the *Web Service Description Language* (*WSDL*) [CCMW01] and extends WSDL's static interface definitions with elements which describe message exchange sequences, including complex constructs like loops or exceptions.

Further highlighting communication within services, Benatallah et al. use FSMs to describe interaction behaviour of service roles in terms of protocol automata [BCT04a, BCT04b]. Within their work, a strong relation between the communication semantics of a service role's interface and primitives of the applied networking middleware is drawn, promoting this dependency as one key element driving the modelling process. Based on the formal specification of interface protocols, they apply model checking techniques to proof compatibility between service role implementations, and replace-ability of components within system specification. Based on protocol simulation, they present a taxonomy to qualify service (role) similarity. As Li et al. pointed out in [LJJ05], a pure behavioural protocol specification only based on state transitions may not be sufficient to proof suitability of services in complex scenarios. In consequence, they presented their work as extensions to the application of FSMs in the context of service modelling. They provide means to semantically extend the formalism by non-functional interaction constraints annotated to individual transitions.

In contrast to the previous work, the following approaches use Petri-Nets [Pet81] for behaviour modelling. The *Radl* framework [LPS03] orients itself on *Architecture Description Languages* (*ADLs*) to describe service oriented system by a process and a structural view. Like ADLs, Radl distinguishes three major concepts: components, referred to as *kens* which come in a basic and a composite flavour, *gates*, representing ports, and connectors. Using Petri-Nets, Radl specifications model the dynamics of the underlying business workflows which are captured in services.

WS-Net [ZCCK04] can be seen as an extension to Radl. By using coloured Petri-Nets [Jen90], WS-Net supports the description of service components on three different layers, allowing a better separation of concerns during system design. The *interface net* is used to model the functionality a component provides. Its counterpart, the *interconnection net* identifies the resources a component acquires from others to accomplish its own task. Finally, the *interoperation net* focuses on the internal behaviour of a component with respect to its interactions with the environment. By embedding appropriate tool support into the execution environment, WS-Net service configurations can be verified and monitored during run-time.

Drawing even more attention to the execution state of service oriented systems, Dijkman et al. [DD04] uses Petri-Nets for service composition and orchestration, i.e., controlling the wiring in-between service components at run-time. Therefore, component oriented aspect, i.e., interface and processing behaviour, are extended with service choreography, i.e., the "storyboard" of service interaction. "By formally capturing the interrelationships between these viewpoints, the proposal enables the static verification of the consistency of composite services designed in a cooperative and incremental manner" [DD04].

In [Krü04, KM04], Krüger et al. presents a methodology for service modelling and composition based on a self defined ADL dialect combined with MSCs [oI01]. Thereby, services are understood as interaction patterns[1] among a set of entities. A service identifies each such entity as dedicated *role*, which plays a well defined part within the captured interaction. The communication semantics between roles of a service are specified with MSCs, describing the externally observable behaviour of a role. In relation to ADLs, roles are implemented by components which, in turn, are then composed to system configurations.

Standalone Formalisms

The category of standalone formalisms for service oriented system design addresses work which is not derived from a separate formal framework. Instead, the authors of the following methodologies invented their own mathematical models. However, most work is at least weakly based on the work of Briand [BMB96] who present an quite generic framework for modular software system design. In its most abstract manner, for Briand, a system S will be represented by a pair (E, R) where E represents the set of elements of S and R is a binary relation between such elements, i.e., $R \subseteq E \times E$. R is understood as expressing links between the system's elements. Briand's model is intentionally kept abstract to serve as a base for formal system models in various applications. Towards the description of service oriented systems, the work of Rossi et al. [RF03] plays a key role as they adopted Briand's concepts for distributed systems. Therefore, they refined the relation R to represent communication between distributed components.

Based on this preliminary work, Perepletchikov et al. [PRFS07] proposed a formal model for service oriented design. Their mathematical model covers essential design artifacts for services for both, structural and behavioural properties. The model is tailored to the *BPEL for Web Services* (*BPEL4WS*) [ACD$^+$03] and is based on directed graphs and set theory. Thereby, software components are represented by vertices and their dependencies by edges. Following their notion of services, a service is a connected sub-graph of the system graph. Communication behaviour is expressed by a proprietary process algebra which can be mapped to BPEL4WS, leading to executable system specifications.

Inspired by the telecommunications domain and the *International Standards Organisation* (*ISO*)/*Open Systems Interconnection* (*OSI*) reference model [Zim80], Herberg et al. developed a layered model for service design. In [HB05] they show how to model software architectures and service interfaces offered and consumed by components that could be arranged in form of layers. In such an architecture, a component representing layer n offers functionality to its upper layer $n + 1$ and can access resources of its lower layer $n - 1$. Consequently, a service describes the interaction in-between two adjacent layers. A typical example of system which can intuitively modelled in this manner is a client/server application.

An exhaustive formal framework for modelling service oriented architectures is presented by Broy et al. in [BKM07]. Service components are described by their structure as well as their behaviour. The collaborative behaviour of services is then a result of a formal merging process. The approach is based on the Focus framework [BS01]. Within Focus, systems are compositions of *interaction components*. A component is understood as a *total behaviour*, i.e., a realisation of a specific functionality. In contrast, a service is only a *partial behaviour*, stating what is required, in terms of component interaction, to establish a requested functionality. Service interaction is modelled via *timed data streams* which represent the message sequences to be exchanged by components. The mathematical foundations of the approach

[1]Note, the term *interaction pattern* as used by Krüger does not represent a templateable modelling element as used in the presented thesis.

allow for validating various properties of the modelled systems, e.g., the absence of deadlocks for service interactions or reachability of individual system states.

2.1.2 Architectural Frameworks

For unification of competing modelling approaches a reference model for service oriented architectures was presented by the *Organization for the Advancement of Structured Information Standards* (*OASIS*) in [MLM$^+$06]. As a reference model, it identifies artifacts and their relations within a service oriented environment. It does not represent a concrete architecture but solely provides generic key concepts. Thus, it is situated at the most abstract level of system modelling. The core element of the reference model is the *service*. The notion of service combines the following three ideas (cf. [MLM$^+$06, p. 8]):

- The capability of an entity to perform work for another entity.
- The specification of the work offered by an entity.
- The offer of an entity to perform work for another entity.

These points imply a distinction between a capability, the ability to bring that capability to bear which includes the willingness to share that capability, and the need for a capability which can not be provided by oneself. In this context, a service provides a mechanism by which need and capabilities are brought together. A service is described at least by the following properties(cf. [MLM$^+$06, p. 12]):

Description. A service must be described in a standardised manner which can be used to register it with a registry from where it can be discovered.

Interaction. What interactions are necessary to establish a service?

Contract & Policy. Under which circumstances can the service be established?

Effect. What is the observable outcome, the result of the service?

Context. Which assumptions are made for a service to operate as specified?

Visibility. Which parts of a system can "see" the service, i.e., can discover and request the functionality a service offers?

The reference model provides "best-practises" to each of these points and gives advices on how they can be implemented.

Extending the elements of the OASIS reference model, further contributions address organisation and run-time management for service oriented architectures, e.g., the one proposed by Papazoglou [Pap03]. Initially, his model identifies common roles in service oriented systems like provider, consumer, and registry. Furthermore, it outlines concepts of service composition, orchestration, and deployment. Additionally it addresses the underlying communication infrastructure with its implications to the system's architecture.

2.1.3 Modelling Services with UML

In the past ten years, model driven service design using UML has gained large popularity. Thereby, contributions mainly focus on the following aspects:

- Defining stereotypes mainly supporting visual enrichment of UML models without providing further semantics to the models.
- Presenting light weight extensions for UML in UML profiles to represent concrete service architectures, e.g., providing specialised stereotypes for specific architectural aspects or a service platform's interaction semantics.
- Proposing heavy weight UML extensions in form of meta-model adjustments reinterpreting UML standard elements in the context of service orientation.
- Describing *Model-to-Text* (*M2T*) transformations and aligned modelling concepts intended to generate service descriptions or implementation artifacts.

Most work in literature does not exclusively address one of these categories in isolation. Instead, often a combination thereof is described. In the remainder of this section, we shortly present related work in the context of UML based service modelling covering representative approaches for every of the mentioned categories.

One of the most popular UML profiles enabling service oriented system design is the one provided by IBM [Joh05]. It defines a number of characteristic, architectural stereotypes following a simple meta-model which relates service providers to consumers, representing the core elements of the meta-model. These elements are connected by services which are described through the messages to be exchanged upon service establishment. As such, it is closely related to OASIS' reference model although not being directly influenced by it. However, the profile's stereotypes have rather decorative semantics as no further modelling constraints or model transformations are defined exploiting the profile. Furthermore, the profile is missing a mechanism to model message exchange patterns, e.g., notification and method invocation. Instead, it just refers to a quite generic message primitive to transmit data between two components.

The same drawbacks apply for the work presented by Heckel et al. In [HLT03] they describe a similar generic UML profile but extended by a special run-time component – the service registry. Although being explicitly designed towards model driven design, they do not demonstrate any model transformations for their approach.

A more detailed and thus expressive UML profile was proposed by Amir et at. in [AZ04]. They defined five sub-profiles each of them devoted to a separate aspect of service oriented designs: resources, service description, exchanged messages, service policies and service implementation. Due to its strict alignment to web services, the model is not suitable to develop system definitions outside the scope of WSDL. Again, the profile is missing an explicit description of interaction semantics.

In [VCM05], Vara et al. describe a design tool for modelling services. The tool is specifically tailored to WSDL and thus supports only the elements defined in WSDL, including the limited set of interaction primitives. The benefit of their approach is that they inherently provide a direct mapping to WSDL documents realised through automated code generation. At the same time, this also represents a disadvantage as the complete tool chain depends on WSDL and thus does not represent a "real" platform independent solution.

In [LGW+07], Likuv et al. present the *UML based Rule Modelling Language* (*URML*). The language focuses on modelling business logic in services. Service functionality is defined via *reaction rules* which consist of four elements: *trigger events* specify under which circumstances a rule becomes active, *rule conditions* define the prerequisites for rule activation once it is triggered, *rule actions* express the concrete effects of a rule, and finally the *post-conditions* express the system state after the execution of a rule. The authors provide a mapping between rules and web services artifacts, i.e., interfaces, operations, and messages. Hence, their system models can be transformed to WSDL descriptions. Like for the previous approach of Vara et al., URML is tailored towards a concrete technical platform (WSDL) which again limits portability.

The *active components* model described by Lopez-Sanz et al. [LSACM08a, LSACM08b] is also a complete meta-model for service oriented systems. Active components realise services by acting as provider, or require services by representing consumers. The relationship between both is defined through *business contracts*. These contracts are modelled by explicit association classes which characterise the connections between components. The meta-model puts special emphasise on user interaction with such systems by defining *front end* components based on active components. Unfortunately, the authors present only the modelling approach itself and discuss no further applications of the resulting models like transformations and code generation.

The *UML Profile for SOA* (*UML4SOA*), described by Koch et al. [KMH+07], is a comprehensive UML profile for service oriented architectures. It is part of a larger project suite called SENSORIA [BFGK06]. SENSORIA is a prototype language for service modelling. Hence, UML4SOA is its adaptation for UML. UML4SOA provides dedicated model elements for structural and behavioural aspects of services and realising components. Additionally, the annotation of business goals, policies and other non-functional properties for services is supported. The approach exploits UML protocol state machines and workflow diagrams to model service interaction and business goals. Service compositions are described as orchestrations via UML activity diagrams. Due to its consequent alignment to UML principles, UML4SOA constitutes the basis for model transformation and code generation techniques. However, the focus is put on modelling business logic applications in form of service compositions which are automatically derived from UML models and executed via appropriate interpreters. There is no explicit support to model service interaction primitives as first class entities and their respective groundings to middleware technologies.

A similar approach is proposed by Emig et al. in [EWA06]. The structure of service based systems is modelled via UML component diagrams, assigning services, i.e., functionality, to components. The functionality itself and thus the dynamic aspect of a system is specified with the *Business Process Modelling Notation* (*BPMN*) [OMG11], a graphical language for business processes. By providing a transformation of BPMN to a combination of *Business Process Execution Language* (*BPEL*) [OAS07] and WSDL, the approach allows automated generation of service components and business flows. Despite the fact of restricting their approach to these two target technologies, the major drawback of this work is that there is no automatic mechanism to ensure coherence between the BPMN and the UML component model of the system. Hence, both models have to be kept in synchronisation manually.

Krämer promotes the interaction between service components as central elements of service orientation instead of the functionality wrapped in components. In [KH06] he describes how one can model services based on UML collaborations for structural and UML activity diagrams for behavioural aspects. The motivation behind his methodology is the description of service compositions. His ideas are furthermore extended by Sanders to support the annotation of *service goals* [San07], enabling automated service composition. Both authors do not provide any model transformations or code generators to support realisations of the specified systems on real target platforms.

Also following the interaction view, Ermagan et al. describe the *Rich Services Profile* [EK07]. Hereby, services are described by component collaborations. The approach addresses on the one hand, the logical architecture of a system as it is decomposed by services. On the other hand, the authors describe the deployment architecture, i.e., the distribution of service components to physical nodes of a system. Their particular focus lies on the controlled aggregation of individual services into composite system architectures. The approach is tightly coupled to WSDL and BPEL.

On the way to an agreed reference profile which unifies the various approaches in industry and research, the *Object Management Group* (*OMG*) initiated a “Request for Proposal” to define a standard UML profile in accordance to the ideas of model driven development [AO06]. The intention of this request is to define a common vocabulary and meta-model for service specification. Although the request’s answer deadline has passed in 2007, there is still no draft of a standard document available from the OMG.

2.1.4 Discussion

When modelling services one currently has to choose between formal frameworks or UML based approaches tailored to specific reference technologies or architectures. Formal frameworks typically provide a mathematical basis to describe the structure of service oriented systems as well as their dynamics. Thus the occurring communication within such system can be explicitly annotated supporting automated validation. However, due to their mathematical origins, such frameworks do often not integrate themselves seamlessly into model driven software development as they omit the relevant modelling entities projecting the mathematical constructs onto software elements. Work resolving this drawback often aligns the formal approaches with very specific approaches of service-oriented computing, e.g., WSDL. Hence, only service oriented systems following a very specific methodology can be modelled.

2.2 Service Interactions

The work discussed in the previous sections mainly covers the overall context of modelling services and systems thereof. Complementary, the following sections delve into the more specific questions about how service interactions are explicitly described and realised.

2.2.1 Interaction Patterns

Component interaction represents an integral part of any software system, may it be monolithic or distributed. “Actually, interactions between communication entities are essential in the description of behaviour.” [Byu03, p. 12] They describe the way how components relate and communicate with each other. Although such communication is a highly dynamic process that is “difficult to partition and categorise” [Fai98], interactions often have basic characteristics which can be resolved by abstraction and documented in a manner that they can be applied to various application domains. The outcome of this extraction process is an *interaction pattern*.

The technical notion of *patterns* originates from the work of the architect Christopher Alexander. When describing his thoughts about fundamental principles of urban architecture design in 1979, he coined the usage of "pattern" to describe "something which repeats itself over and over again, in any given place, always appearing each time in a slightly different manifestation" [Ale79, p. 181]. Clarifying the "something", Alexander explains that a pattern is a three-part rule which correlates a problem in a certain context to a specific solution [Ale79, p. 247]. By providing an intuitive vehicle to describe expert knowledge in a general-purpose fashion [vdBC01], the concept of patterns was rapidly adopted to the field of software engineering. Thereby, a number of catalogues were created mainly addressing software design problems as presented in, e.g., [GHJV95] or [BMRS96].

Interaction patterns describe proved solutions of modelling interaction within the context of communicating entities. They capture the assumptions, expectation, understandings, and goals that drive components to communicate. Thereby, the following aspects are of special interest (cf. [Fai98]):

Roles. What are the entities/processes to communicate with each other? Which is providing what information or resource?

Control. Who is in charge of the interaction? Which process initiates the interaction and which one terminates it?

Timing. What dependencies exists with respect to time and execution progress? Does one process wait for the other?

Flow. How is information transmitted between the participants? Is it represented by an atomic message or split to multiple ones? How do messages relate to each other?

Acting as the glue between components upon system integration (cf. [Esk99]), structure and behaviour are often layered on top of interaction patterns. In literature exist many approaches to not only present catalogues but also taxonomies of interaction patterns. The range lasts from the fundamental classification of forms of communication, to focusing on the peculiarities of the interaction roles. The former category addresses aspects of synchronous and asynchronous communication as presented by Tanenbaum and Steen in [TvS01]. The latter, describes patterns like provider-consumer, stating dependencies in terms of data creation and consumption, or push-pull, expressing if the producer of data or its consumer triggers the data flow (cf. [Fai98]). For a more detailed discussion on such catalogues please refer to Section 2.2.2 on page 17.

Summarising, interaction patterns as used in the presented work, are defined as following:

Definition 1 (Interaction Pattern)**.** *Interaction patterns generalise a common sequence of actions, i.e., message exchanges, occurring in interactions between a pair of communicating entities.*

Thus, the patterns describe similarities of interactions in terms of interfaces and semantics. They provide a vehicle to identify and capture interactions and making them available for re-use. Not only the structural equivalence of interactions, but also their intention, or semantics, should be captured in a pattern. Interaction patterns identify dependent sequences of message exchanges and thus represent building blocks to be re-used through composition to specify complexer, higher-level communication models, i.e., services.

2.2.2 Catalogues for Service Interaction

There exist many different schemes of interaction to be used within services. Hence, several authors tried to give classifications and present taxonomies to categorise them. We present selected work of this field of research and describe their underlying principles, omitting the reproducing of the complete catalogues.

Faison described an interaction catalogue in [Fai98], addressing bilateral interactions, i.e., interaction between only two entities. He identified fundamental properties which characterise an interaction, e.g., synchronous vs. asynchronous communication, and monitorability or abortability of the actions caused by a communication.

In contrast to this work, Eskelin [Esk99] classified interactions based on the form of organisation/inter-connection. Thus, he mainly addressed questions about the communication infrastructure rather than about their semantics. In his words "assembling a system consisting of custom and pre-built components can be difficult because of hidden dependencies, complex interactions, and obscure design" [Esk99]. In consequence, he proposes five architectural patterns to simplify component assembly in terms of communication, collaboration, and coordination. The patterns are *abstract interactions* as black box of communication itself, *component bus* as centralised communication infrastructure, *component glue* to connect components to the bus, *third-party bindings* to mediate between different technologies, and *consumer-producer* to model data flow.

Barros et al. [BDtH05] focus on the number of participants for an interaction. They present a collection of patterns which cover combinations of single vs. multiple sender, single vs. multiple receivers, and different variants of message routing between both. The described communication semantics originate from the BPEL/WSDL context and are investigated under special consideration of their implications to service choreography and orchestration.

Finally, Mahfouz et al. [MBLN06] address the handling of interaction constraints beyond pure communication/invocation semantics. "The patterns aim at explicating and elaborating the business requirements driving the interaction and separating them from implementation concerns" [MBLN06]. Beside others, the discussed patterns comprise barriers, deadlines, solicitation, queries, and expiration.

Apart from the work of Barros et al. the catalogues are represented by abstract pattern languages describing *best practices* for interaction modelling depending on the concrete context of communicating entities. Hence, they serve as starting points for modelling activities, i.e., the patterns have to be adapted into formal models prior to their exploitation within a service modelling process. When focusing on Barros et al., they address a very specific subset of service modelling activities with respect to BPEL and WSDL. Thus, there work also need some adaptation to support the more generic notion of services as proposed by the this thesis.

2.2.3 Modelling Service Interactions

Service interaction can be described by modelling the communication protocols which connect the service participants. Within this section, we distinguish between pure formal and UML based approaches.

Formal Approaches

A representative approach based on FSMs is described by Byun [Byu03, BS05]. He describes a pattern language for the design of communication protocols. Within his methodology, orthogonal characteristics of a communication, e.g., message flow or time constraints, are specified by a number of independent communication artifacts in form of FSMs. A concrete interaction is then defined by a composition of such artifacts by merging the associated FSMs. Byun then uses automated validation checks on the resulting automata using PROMELA.

Kazhamiakin et al. developed an analysis framework for service communication and composition [KPS06]. Their work focuses on describing interaction in business processes. Therefore, they provide a formalisation of BPEL via FSMs and use it to model basic synchronous and asynchronous communication. Based on this formalism, they investigate the behaviour of composite services by combining the FSMs of the underlying primitives.

Decker et al. [DPW06] present a comparison framework for two other popular formalism for protocol design, i.e., the π-calculus [MPW92] and Petri-Nets [Pet81]. They first demonstrate how typical interaction patterns, they use the ones defined by WSDL, are formalised in both formalisms. They investigated different approaches using various types of Petri-Nets as well as a multiple extensions to the π-calculus.

The work of Zaha et al. [ZDtH$^+$06] is a representative for approaches using self-defined, dedicated mathematical models. The authors present their own formal grounding on set theory and provide a textual as well as a graphical method to describe service interactions and choreographies. Hereby, choreographies are seen as the *global* view on service oriented systems, individual interactions as the *local* view. Within their approach, one directly starts by specifying detailed choreographies for services. A service itself is intentionally *underspecified* with respect to its communication interfaces and protocols. This missing information is automatically derived from the choreography specification and results in varying behaviours for each of the actors which participate in an interaction depending on the system context.

Benatallah et al. developed a meta-model and framework to define service *conversations* [BCT04b]. A conversation describes the set of acceptable message exchanges, i.e., message types and their order, in an FSM like language. The authors analysed e-commerce portals based on WSDL and extracted a number of typical conversation patterns. The patterns are classified into two categories: *completion* and *activation*. Completion patterns describe a transition's implications and effects from the requester's perspective. For instance, a pattern captures the possibility to cancel an operation by the caller. A completion pattern is described by the following properties: its persistent *effects* on system state, its *compensatability*, i.e., if its effects can be undone, *retriability*, i.e., its idempotence, its *credential-disclose* stating what state information is evaluated to establish the conversation, and its *locking* semantics on system resources. In contrast to completion, an activation patterns describes the features which trigger an interaction, e.g., by explicit invocation or on fixed schedules. Hence, an activation is described through the *trigger conditions* as well as *temporal* constraints.

UML Based Approaches

Castejón et al. describe a method to model service interactions with UML collaboration and activity diagrams [CB06]. Grounding on these models, they present an analysis framework to determine *implied interaction* scenarios which result as side effects when composing services to a complex system. With the evaluation of the implied interactions, they validate *well-*

formedness of the overall system behaviour. The authors do not address the derivation of interaction realisations on concrete middleware technologies, may it either be manually or automatic.

Kramler et al. [KKRK06] model service interaction on two different levels of abstractions. First, the *interaction level* considers transactions and transactional processes between communicating entities. Each transaction is performed by a set of participants of a collaboration, referred to as service. This level focuses on a single atomic sequence of message exchanges occurring as building blocks of component interaction. Next, on the *collaboration level*, they model state progress and control flow between the service participants. They present a graphical representation based on UML collaborations, state machines and sequence diagrams. As their approach is closely aligned with BPEL, they do not consider generic mappings of modelled interactions to varying target platforms.

Another approach based on UML collaborations is described by Krüger et al. [KM04]. In their view, “a service is defined by the interaction among the entities involved in establishing the service” [Krü04]. Therefore, they exploit UML collaboration diagrams to model structural relations of service participants combined with MSCs describing message exchange. The authors provide the concepts of *sequences*, *alternatives*, *repetition*, *interleaving*, and *joining* to compose service protocols based on the individual communication behaviour of each service participant. The presented approach does not support automatic protocol composition as some concepts, especially interleaving and join, require expert knowledge from the service designer. Furthermore, Krüger does not discuss how the resulting protocol specifications can be realised on concrete platforms.

Birkeland [Bir06] also presents a generic modelling approach for service interactions using UML collaboration and sequence diagrams as well as and protocol automata, i.e., FSMs. He describes individual *aspects* on protocols as UML collaboration templates which can be applied to varying service scenarios via instantiation. Thereby, a template directly captures communication behaviour on the service protocol layer. Hence, his approach does not identify the underlying interactions of such protocols as stand-alone primitives for service modelling and key elements to middleware groundings. Instead, Birkeland focuses on questions about the composition of FSMs as a result of template instantiation for service definition. Such composite FSMs are then validated with respect to, e.g., state reachability and termination.

2.2.4 Service Adaptors

Service adaptors are used to realise service interaction on concrete middleware platforms. They fill the gap between the interaction platform independent service specification and the service's execution in a real system environment. Furthermore, adaptors can be applied for achieving compatibility between mismatching role implementations within a common service.

Pham et al. [PCS07] present a formal model for protocol adaptation based on specification pattern systems. Using a pattern library, they create FSMs which mediate between previously incompatible service role interfaces. By providing a complete run-time environment for their approach, they can modify these mediators during system execution to react on changes in the system infrastructure, e.g., the replacement of service role implementations.

Similar approaches are described by Gierds et al. [GMW08] and Li et al. [LFW^{+}08]. Both ground their work on set theory and Petri-Nets. Service adaptors are defined with so called *specifications of elementary activities* or *protocol mediators*, respectively, which describe sets of message transformations rules which must be applied to mediate between two service role interfaces. The rules include *creation* patterns which are used to inject additional control flow messages if necessary, *copy* for message repetition, *delete* to suppress messages, and *transform* to reformat messages.

Beside establishing compatibility on the *service protocol level*, there exist works in literature which also addresses the technical grounding of services on the *operation level.* Benatallah et al. [BCG$^+$05] presents one such approach. On the service protocol level they also exploit pattern based solutions for message reordering, dropping and creation. Furthermore, they consider the influence of the communication middleware to the service adaptors, e.g., available message exchange primitives, and weave them into the adaptor specifications. As described by Nezhad et al. [NRB$^+$07], the approach can be extended to support at least semi automated resolution of protocol mismatches and the generation of service adaptors in the context of WSDL.

Ihmor et al. [IH05] describe an approach to adapt varying inter-module communication protocols based on data in-/outboxing and protocol translation. Although, the addressed application context lies within dynamic hardware reconfiguration, the protocol translation concept seems also applicable to software protocol stacks as used in service oriented or middleware-based computing.

Nakazawa et al. [NTER06] developed an adaptive middleware, referred to as *uMiddle*, which allows flexible groundings for services. Their framework supports protocol bridging and data type conversion via a plug-in architecture. The *M²* [YZCM04] framework uses an analogous technique. *M²* augments service components via *transparent shaping*, i.e., it uses configurable service adaptors as communication back-ends which can be adjusted at run-time.

The major drawbacks of the presented approaches are on the one hand that they often include manual design steps as in practice not all protocol mismatches can be resolved automatically. On the other hand, these approaches make explicit assumptions about the fundamental principles to be used for communication and thus can not be easily adopted for different target platforms. Using a homogeneous communication back-end at least solves these problems with respect to middleware adaptation. For service oriented architectures, the concept of an *Enterprise Service Bus* (*ESB*) [SHLP05] represents the de facto standard solution. An ESB uses *service containers* to connect arbitrary service components to a unified messages systems [Cha04, p. 58]. More clearly, "an Enterprise Service Bus is an open standards, message-based, distributed integration infrastructure that provides routing, invocation and mediation services to facilitate the interaction of disparate distributed applications and services in a secure and reliable manner" [Men07]. Due to its simple architecture the concept is currently widely used for commercial products, e.g., IBM WebSphere Enterprise Service Bus[2], Progress Sonic ESB[3], and Oracle Enterprise Service Bus[4].

In contrast to the ESB, our approach realises service interaction based on generic interactions for services which are natively mapped to each platform's primitives on a per interaction basis. Hence, we can avoid further indirections of component communication if a selected middleware directly matches the interaction semantics. Furthermore, we allow subsets of interactions within the same service to be mapped to different technologies, e.g., based on native support for some interaction characteristics like *Remote Procedure Call* (*RPC*) or

[2]Please refer to `http://www.ibm.com/software/integration/wsesb/`.

[3]Please refer to `http://web.progress.com/en/sonic/sonic-esb.html`.

[4]Please refer to `http://www.oracle.com/technology/products/integration/esb/index.html`.

Requirement	Service Modelling			Interaction Modelling				
	Form.	Arch.	UML	Patterns	Catal.	Form.	UML	Adapt.
Pattern Lang.	–	–	–	+	+	o[a]	o[a]	o[a]
Form. Inter.	+	o[a]	o[a]	o[b]	o	+	o[c]	o
Form. Serv.	+	+	o[c]	–	–	+	o[c]	–
Form. Sys.	+	+	o[c]	–	–	o	o[c]	–
UML Profile	o[da]	o[d]	+	–	o	–	+	o[d]
Adaptor Gen.	–	–	o	–	–	–	o	+
MDA Process	o[d]	o[ad]	+	–	–	–	+	+

Table 2.1: Requirements coverage matrix ("+" – fulfilled, "o" – partially fulfilled, "–" – not fulfilled).

[a]Implicitly defined by fixed set of interactions.
[b]Semi-formal/textual description
[c]Implicitly defined by UML semantics.
[d]Partially supported as part of intetegration into development process.

message passing. Such combinations of target platforms are difficult to realise with ESB or similar architectures as they assume one standard platform as communication back-end.

2.2.5 Discussion

Current work in the field of explicit modelling of service interactions, one has to distinguish between two main streams. The first stream contains approaches to describe interactions in a quite generic manner, i.e., pattern languages. These works provide sophisticated solutions to capture interaction semantics but often do not draw the connection to their application in service development processes, especially with respect to MDA based methods.

In contrast, the second stream especially considers the integration of service interactions into such development processes. The major advantage of these approaches is their inherent support for model driven development processes including code generation and system deployment via automated groundings of service interactions to specific platforms. The drawback of these works is their typical strong alignment to very specific service platforms or communication middleware. They do often not support the modelling of arbitrary service interaction as first class modelling entities as they rely on fixed sets of communication primitives supported by their underlying technology.

2.3 Discussion

Table 2.1 shows the overall requirement coverage matrix of the discussed work with respect to the list of requirements as identified in Section 1.3. A requirement is fulfilled if work of the respective category provides an explicit solution to that requirement, or it is not fulfilled if it is not addressed. A partially fulfilled requirement results from only limited support by the work of the corresponding category.

Work of the field of service modelling provides generic solutions for formal modelling of services and their interactions. But focusing on their support for MDA/UML based processes, they show strong limitations as they typically concentrate on very specific subsets of service interaction semantics. In turn, if provided at all, they do support the automatic generation of service groundings (through adaptors) only for that subset.

Research in the field of interaction modelling for services draws attention primary on two points: categorising interactions by use of pattern languages and catalogues or providing means to generate service groundings. Due to the generality of the first, they provide support of describing a variety of interaction mechanisms but omit the transformation thereof to concrete technologies. This transformation if the strength of the second group of related work which offers support for automatic groundings but it turn uses closed sets of interaction patterns as a basis as for related work in the field of UML based service modelling.

What is missing is a holistic development process which covers all the requirements. A process which enables the development of an extensible interaction pattern library which can be directly used for service and system modelling with automated generation of service groundings. A process that supports both, an MDA/UML based modelling complemented by appropriate formal semantics of the modelled elements.

CHAPTER 3

Model Driven Architecture

This chapter gives a short overview of MDA as it forms the basis of the proposed development process. It presents a summary of the underlying methodology as well as of its both core concepts, models and transformations.

3.1 Methodology

MDA is a system design and development methodology to assist system development with an technology independent approach. MDA was first described by the OMG in 2001 [OMG01]. One of the main goals of the OMG is to establish open, vendor-neutral and thus interoperable specifications for system design and integration. In that sense, MDA embodies the vision of presenting a holistic framework to support interoperability with specifications throughout a system's complete life cycle. The design philosophy behind MDA shall address the description of a system's business logic, modularisation, component construction and integration, as well as deployment, management and evolution [OMG01].

A key concept of MDA is the separation of the specification of functionality from the specification of implementation, integration and deployment [OMG01, MM03]. This is achieved by applying *abstraction* to the design process. Hereby, abstraction is understood as the suppression of irrelevant detail as motivated by the reference model for open distributed processing [FLdM96]. Abstraction's counterpart in MDA is *specialisation*, also referred to as *refinement*. Once a system's functionality is defined on an abstract level, this definition is enhanced by more and more implementation detail until the system is specified on its implementation level.

Through this architectural separation of concerns within a system's life cycle, MDA addresses three primary goals: portability, interoperability and re-usability. Therefore, the OMG provides concepts and tools through MDA to specify a system independently of the addressed deployment platforms, to model these platforms themselves, to choose a particular platform for the designed system out of multiple candidates, and to transform the system specification into one compatible with the selected platform [MM03].

Note, MDA itself does not imply a concrete development methodology. It only prescribes

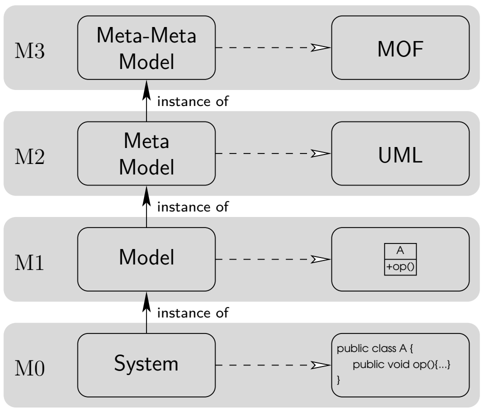


Figure 3.1: The Modelling hierarchy (derived from [OMG05a]).

a framework and abstract tools to support a model oriented system design, evolution, and maintenance process. MDA does not enlightens design activities, roles, or phases in terms of a guidance to the design process [GBPA04]. As a result, albeit MDA compliant, individual projects define their own or adapt existing methodologies.

3.2 The Model in the Model Driven Architecture

The fundamental element in MDA is the *model* as the term Model Driven Architecture already emphasises. Within MDA, a model refers to "a representation of a part of the function, structure and/or behaviour of a system" [OMG01]. This representation may describe arbitrary specifications of a system's structure, behaviour, or environment and may be presented as text or drawings. As such, a model is a formal description of a complex system or application artifact [PB03].

A typical system specification consists of many models. Thereby each model reflects a different view or subsystem. To guide the creation of models, MDA provides two approaches to a developer. The first concept, model *refinement*, provides means to add more detailed information about the system to an existing model. This information may expand formally abstracted aspects of the system or may add implementation specifics [OMG01]. Model refinement can be seen as an in-place operation where the model itself is modified. In contrast to that, the second concept, model *transformation*, will create new models based on existing ones. By applying model transformations, model elements of an existing model will be converted or *mapped* to elements of a newly created model. Model transformation will be discussed in more detail in Section 3.3.

Using the concepts of models, refinement, and transformation, MDA accompanies a system's full life cycle by providing means for using models to specify the evolution of systems – implying a well defined syntax and semantics of these models [MM03].

3.2.1 The Modelling Hierarchy

Regardless of the form of annotation, e.g., textual or by drawings, a model is described by meanings of a concrete syntax and semantics. This syntax specification can be seen as a model itself – a model of a model – referred to as *Meta-Model*. For MDA, the OMG has identified four layers of modelling [OMG05a], referred to as *M3*, *M2*, *M1*, and *M0*. These layers are depicted in Figure 3.1. The topmost, and thus most abstract layer, is *M3* – the *Meta-Meta-Model* layer. Such a meta-meta-model defines the most fundamental primitives and concepts which are necessary to describe a model. In the context of MDA, this layer is described by the *Meta Object Facility* (*MOF*) [OMG06]. The MOF is a meta-data modelling and management framework to be used to specify modelling languages themselves. Such modelling languages are represented in the next lower layer *M2* – the *Meta-Model* layer. A meta-model represents an incarnation of a domain specific language. A prominent example of such a meta-model definition is the UML [OMG09a, OMG09b, OMG07b] which is discussed in more detail in Chapter 4. Based on meta-models, a system designer specifies a system models in layer *M1* – the *Model* layer. Finally, on layer *M0* – the *System* layer, the concrete system is described as an implementation complying to the model in layer *M1*.

3.2.2 Platform Independence of Models

MDA classifies models into four categories depending on a model's relation to a *platform*, providing different levels of abstractions of the modelled system. In terms of MDA, a platform is defined as "a set of subsystems and technologies that provide . . . functionality . . . which any application/system supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented" [MM03].

The following descriptions of the four categories are based on [MM03]:

Computation Independent Model. The *Computation Independent Model* (*CIM*) is a view of the modelled system focusing on the role it plays within the system environment. The CIM hides any structural and behavioural information of internals of the system. The CIM is also referred to as domain model or business model.

Platform Independent Model. A *Platform Independent Model* (*PIM*) is a view of the modelled system presenting information about the system's structure and/or behaviour but omitting any detailed knowledge about how its low level functionality is being realised by the supporting platform.

A PIM of a system may be seen as a system realisation based on a virtual machine. Although the virtual machine itself represents a platform, and though the model would not be platform independent, the virtual machine hides details about how its own functional primitives are realised on a real computing platform.

Platform Model. A *Platform Model* contains modelling entities describing the individual functional parts of a particular platform, such as *Application Programming Interfaces* (*APIs*), functional building blocks, or technical concepts.

Platform Specific Model. In contrast to the PIM, a *Platform Specific Model* (*PSM*) is a view of the modelled system including detailed information about the platform the system is realised on. The PSM combines the PIM and the Platform Model. Thus, the PSM describes knowledge how the system uses a particular platform and how it is mapped to the platform's building blocks. For instance, if a PIM specifies the usage of an abstract ordinal type for number representation, its related PSM may use a 32 bit integer type.

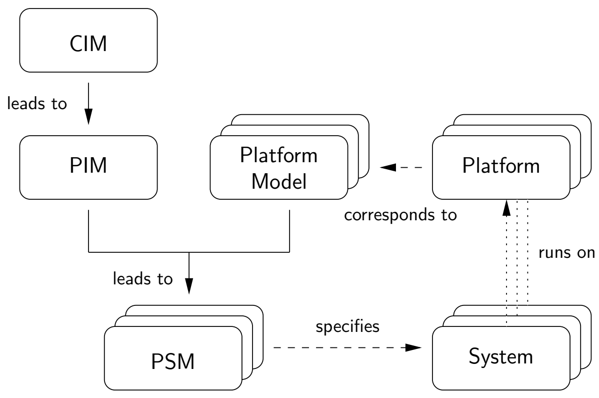


Figure 3.2: The four model categories and their relation to system and platform.

Figure 3.2 depicts the relationship between these four model categories and their association to platform and system. The CIM provides the most abstract view to the system as seen from the outside. This view is extended by a functional separation of the system as given by the PIM. As a parallel modelling activity Platform Models are specified reflecting the platforms' specific characteristics. Combining the PIM with a Platform Model results in a corresponding PSM which now contains a detailed, deployable system description realising the CIM specification on a particular target platform.

Note, platform independence is a quality of a model and is always defined relatively as already indicated above. Assume a high-level model of a distributed system given by a PIM. This system shall be implemented based on the *Common Object Request Broker Architecture* (*CORBA*) [OMG08a, OMG08b]. In the notion of MDA, CORBA can be seen as a concrete platform given by an adequate platform model, e.g., [OMG02b]. For instance, by applying a model transformation, one can turn this PIM into a PSM, thereby platform dependence is relative to CORBA. However, in the next design step, an implementation for the CORBA middleware has to be chosen. In this second step, the previously generated PSM can be seen as a PIM again, because now, the next PSM would represent a the modelled system bound to a concrete implementation of a CORBA middleware, e.g., the ACE ORB[1].

3.3 Model Transformations

A model captures a concrete system design at a given point in the system's life cycle, providing a specific level of abstraction of system structure or behaviour. A system design evolves by model refinement activities, thereby more knowledge and details about the system under development are added to the system's models. Complementing these modelling tasks, *Model Transformation* is the second design activity in MDA. As Kleppe et. al [KBW03] define, "a [model] transformation is the automatic generation of a target model from a source model, according to a transformation definition." Thus, a model transformation is used to *automate* design activities, most often to derive a PSM from a PIM. By applying model transformations, a system's model advances iteratively from an abstract, technology independent model to a concrete, platform dependent one. Eventually the system's (source) code and deployment information are generated as final result of the MDA design process.

[1]Cf. `http://www.theaceorb.com`.

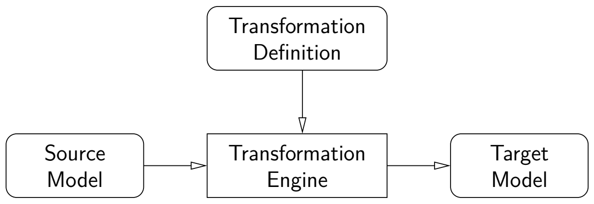


Figure 3.3: The basic model transformation chain (derived from [KvdB03]).

Figure 3.3 illustrates the previously presented definition of model transformations. It relates four entities: a *source model*, a *target model*, a *transformation definition*, and a *transformation engine*. The source model on the one hand is the initial point of a transformation. It represents the source design which is going to be modified by the transformation. The target model on the other hand is the end point of the process, reflecting the result of the transformation. The most important entity is the transformation definition. This definition, interpreted by the transformation engine, describes the rules that select elements from the source model and how these map to elements on the target model. Therefore, a transformation is also referred to as *mapping* [MM03]. The source model may be annotated by meta information prior to being applied for a transformation. Such a *marking* [MM03] indicates special handling of source model elements which may not be expressed by the transformation rules themselves.

3.3.1 Source and Target Models

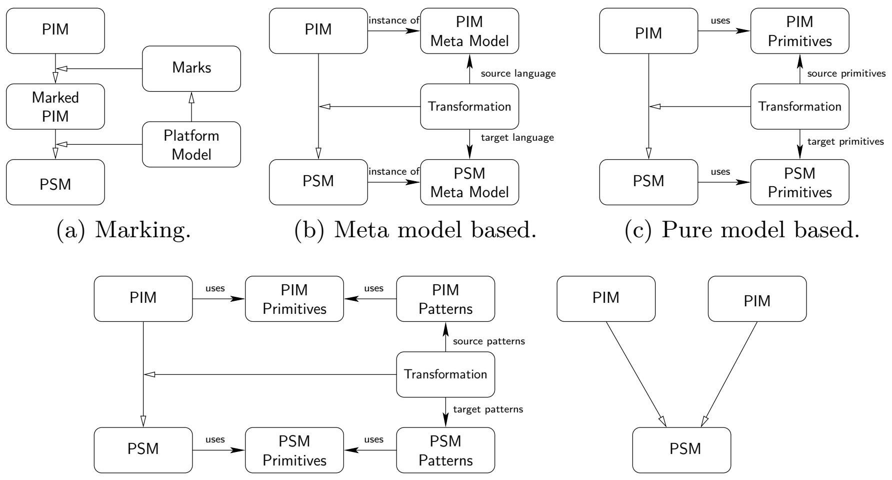
According to [MM03], a model transformation effects PIMs or PSMs. Thus, four different combinations of source and target models are possible:

PIM to PIM Transformation. PIM to PIM transformations are related to model refinement. This transformation is applied when a model is filtered, adjusted, or expanded without adding further platform relevant information.

PSM to PSM Transformation. PSM to PSM transformations are used for modularisation and deployment of run-time components. For example, processes are distributed and assigned to independent computation nodes of the target platform. PSM to PSM transformations can also be seen as a variant of model refinement with respect to platform characteristics. In contrast to PIM to PIM transformations, neither the system's structure nor behaviour definitions are affected by this kind of transformation.

PIM to PSM Transformation. A PIM to PSM transformation is the most relevant form of transformations in MDA. Hereby, a model, being free of platform dependent knowledge, is projected to its execution infrastructure. This projection is based on the platform characteristics given by a platform model.

PSM to PIM Transformation. PSM to PIM transformations represent some kind of re-engineering tasks. A concrete system is analysed and its platform dependencies are removed by abstraction. This is the inverse transformation of a PIM to PSM transformation.



(a) Marking. (b) Meta model based. (c) Pure model based.

(d) Pattern application. (e) Merging.

Figure 3.4: Transformation strategies (cf. [MM03]).

3.3.2 Transformation Strategies

Independent of the kinds of source and target models for a transformation, the OMG has identified several approaches to be applied for model transformations [MM03] (cf. Figure 3.4):

Marking. For marking, elements of the source model are annotated with special meta tags. Upon execution of the model transformation, these elements of the source model are selected by their marks to control their mapping to elements of the target model (cf. Figure 3.4a).

Meta Model Transformation. For meta model transformations, the transformation rules are defined based on the meta models of the source and target model respectively. By defining the transformation rules based on meta models, the transformation becomes generic in that sense, as it can be applied to arbitrary source models (instances of the source meta model) to generate target models (instances of the target meta model) without the need for special preparations of the source models (cf. Figure 3.4b).

Pure Model Transformation. Pure model transformation is used when elements of the source model are projected to elements of the target model based on a dedicated mapping of elements between source and target model. In contrast to meta model transformations, types from the source model are mapped directly to types of the target model instead of mapping meta model concepts (cf. Figure 3.4c). For instance, primitive data types in the source model are directly substituted by corresponding types for the target model.

Pattern Application. Pattern application is applied if the source model reflects a specific system design pattern and the target model natively supports the realisation of such a pattern. Then all elements of the source model forming that pattern are mapped as a group to the pattern's realising entities of the target model (cf. Figure 3.4d).

Model Merging. Model merging can be used if two or more models of a system have the same level of abstraction but address orthogonal aspects of the system. Then pattern merging provides a way of combining these aspects in just one model (cf. Figure 3.4e).

3.3.3 Transformation Languages

In order to execute model transformations in an automated manner, a formal language is needed to specify the necessary mapping rules. There exists many possible approaches for this task. For example one may use a general purpose programming language to implement an application which executes a concrete set of transformation rules. But such a solution is disadvantageous since it is not generic enough to support varying model transformations. To address this problem, there exist domain specific languages, e.g., the *Atlas Transformation Language* (*ATL*) [BDJ$^+$03], which allow the description of arbitrary transformations and provide appropriate tool support. Because of the manifold existing transformation languages, see Gardner et al. [GGKH03] and Czarnecki et al. [CH06, CH03] for an in depth discussion, the OMG as the leading organisation for MDA currently standardises a transformation language, known as the *Query/View/Transformations* (*QVT*) language [OMG05b]. In 2001, the OMG issued a QVT Request for Proposal [OMG02a] to encourage contributions from scientific and industrial institutions. The final proposed QVT standard will be the result of this unification process (cf. to the standard proposals [Alc03, DST04, KBC04, Wil03, Pat04, QVT03]).

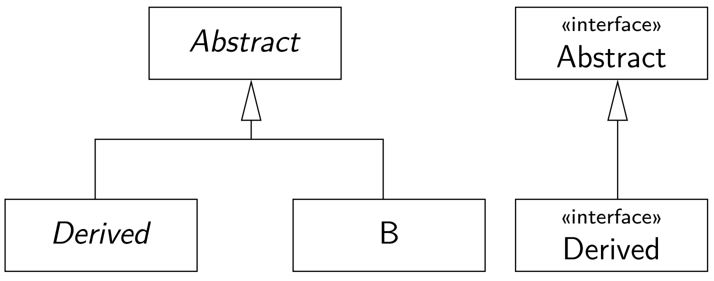
As this unification process is still ongoing, the presented thesis relies on ATL to demonstrate *Model-to-Model* (*M2M*) transformation. ATL is primarily a declarative language but also supports imperative programming to ease development of transformation constructs which can hardly be expressed otherwise, e.g., loops. Preferably using the declarative approach, transformations are described in two parts: a query part, or *source pattern*, which selects elements from the source model, and a *target pattern* part which defines the elements to be created in the target model based on the matched elements of the query part. Listing 3.1 shows an exemplified transformation in ATL. The transformation can be applied to UML class diagrams. It converts abstract classes to interfaces, preserving possible generalisation/specialisation relationships between them. The transformation takes one input model and creates one output model, both based on the UML meta-model (line 2). One rule is sufficient to gain the requested effect. It queries for class definitions in the input model, filtering only these classes which are tagged abstract (lines 5–7). For each match, an interface is created in the output model, having the same name and generalisations as the original class (lines 8–11). The effect of the sample transformation is depicted in Figure 3.5. Only the Abstract and Derived classes are affected by the transformation, as B is not an abstract class in contrast to the other two.

A special category of model transformation is M2T transformation, e.g., as provided through the *MOF Models to Text Transformation Language* (*MOF-M2T*) [OMG07a]. Thereby, the target meta-model is represented by a grammar or other specifications of textual languages. Typically, M2T transformations are applied to generate source code from a PSM, assisting system implementations. Often, the target languages are not explicitly supported by a dedicated meta-model for each language. Instead, transformations are described by templates already expressing valid documents in the target language. These documents are annotated by special tags. The tags contain queries to the source model and are textually replaced by the corresponding matches. Listing 3.2 shows a MOF-M2T template which transforms UML interface models to Java code. The template is laid out individually for

```
module Example;
create OUT : UML from IN : UML;

rule AbstractToInterface {
  from class : UML!Class (
    class.isAbstract
  )
  to interface : UML!Interface (
    name <- class.name,
    general <- class.general
  )
}
```

Listing 3.1: An exemplified M2M transformation using ATL.



(a) Input model. (b) Output model.

Figure 3.5: The models of the ATL example transformation.

```
[template public interfaceToJava(i : Interface)]
public interface [i.name/]
{
  //generate code for operations
}
[/template]
```

Listing 3.2: An exemplified M2T transformation using MOF-M2T.

every interface from the source model which will be bound to the template's parameter `i`. Through this parameter, an interface's properties, like its name, can be accessed from within the template.

CHAPTER 4

The Unified Modeling Language

The following chapter provides a short overview of these parts of UML which are extensively used by the presented interaction-centric service modelling process, i.e., a specific subset of diagram types, templates, and UML profiles. For a more detailed introduction to UML refer to Appendix A.

4.1 Component Diagram

4.1.1 Description

Component diagrams are used to model distinct parts of a system as *components*, i.e., a modular unit with well-defined interfaces [OMG09b, p. 143]. The UML notion of a component is aligned to component-based software development, whereas a component is considered an autonomous unit within a system. By restricting external interaction of a component to a fixed set of interfaces, a component becomes easily replaceable and re-useable. Systems are assembled by combining appropriate components and connecting their interfaces. A component provides functionality to the system through its *provided interfaces*. In turn, the component may depend on functionality being realised not by the component itself but other components by defining *required interfaces*. Interfaces may be grouped by *ports*, representing named sets of interfaces, covering both, provided and required ones.

Where ports and interfaces represent the external, black-box view of a component, it also has an internal, white-box view. A component references *realisations*, which implement the component's behaviour. Such realisations may be provided by a classifier playing the component's role as a whole, or the component is a composite structure (see Section 4.2), realised by subsystems, being components themselves.

4.1.2 Visualisation

Component are shown by a rectangular boxes tagged by a component icon in the upper right corner, the «component» stereotype, and the component's name (cf. Figure 4.2). A port,

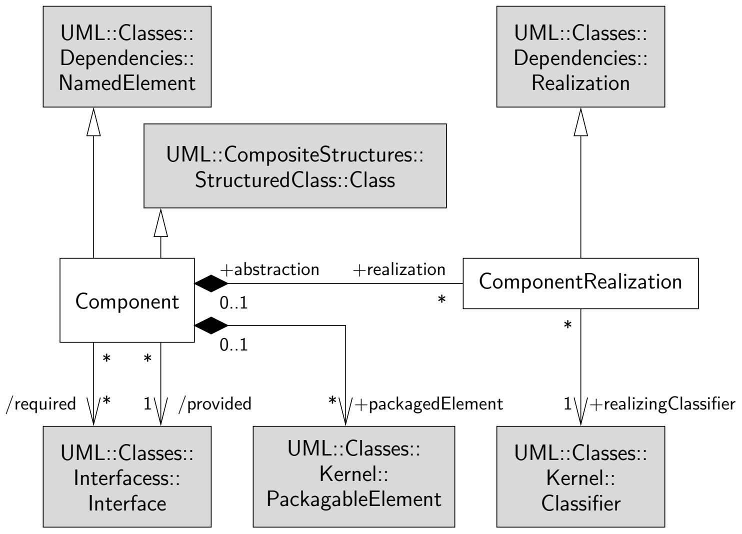


Figure 4.1: The UML component meta-model (cf. [OMG09b, pp. 145–146]).

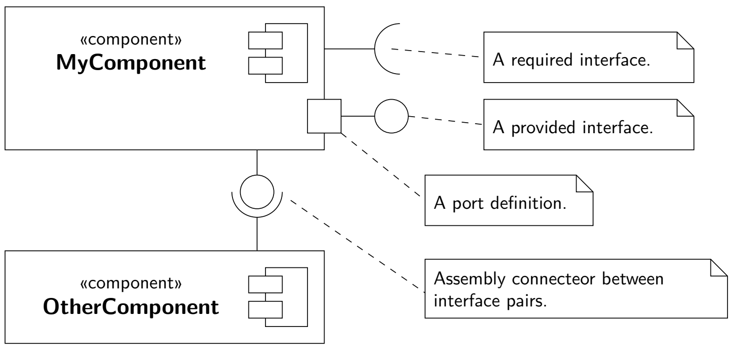


Figure 4.2: A UML component diagram.

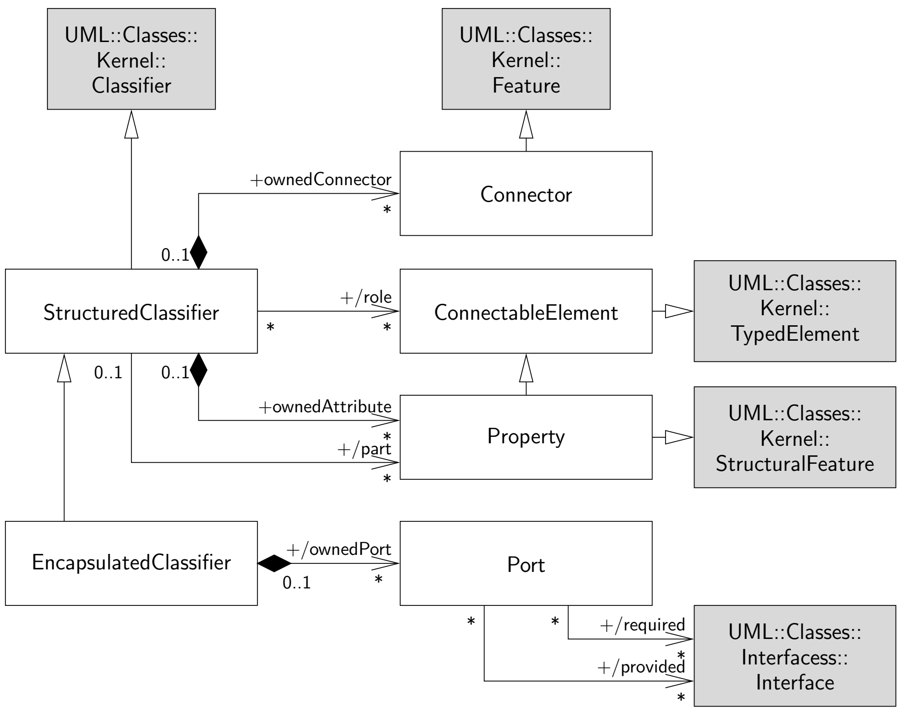


Figure 4.3: Simplified meta-model of UML composite structure (cf. [OMG09b, pp. 163–165]).

attached to a component, is drawn with a box overlapping the border of the component's surrounding box. Ports may be named explicitly by presenting a label close to its box.

Provided interfaces are drawn as “lollipops” sticking out of the component. Required interface use the “socket” notation respectively. Interface names are presented by labels next to the ends of “lollipops” and “sockets”.

A system composition is given by the components itself, often seen as black-boxes, and the connections of required to compatible provided interfaces/ports. Therefore, “lollipop” ends are drawn enclosed by “sockets”. For alternative representation options of components, please refer to [OMG09b, pp. 149–153].

4.2 Composite Structure Diagram

4.2.1 Description

Composite structure diagrams show the internal wiring or statical interplay of parts of a system. The diagrams provide mechanisms for structural decomposition of elements. One prominent modelling concept of this diagram type is the notion of ports. Ports present a way to isolate classifiers from their environment by providing dedicated points of interaction between its internals and the external environment. Ports provide means for complete encapsulation of classifiers so they become easily reusable or replaceable.

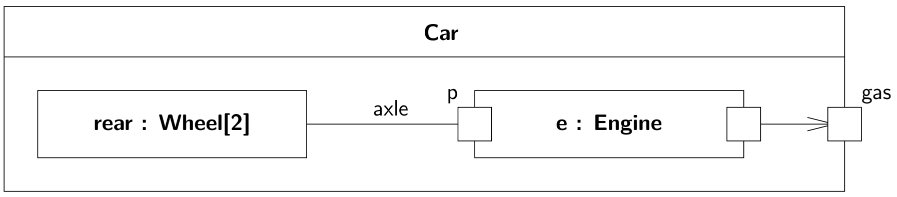


Figure 4.4: Example of composite structures (cf. [OMG09b, p. 183]).

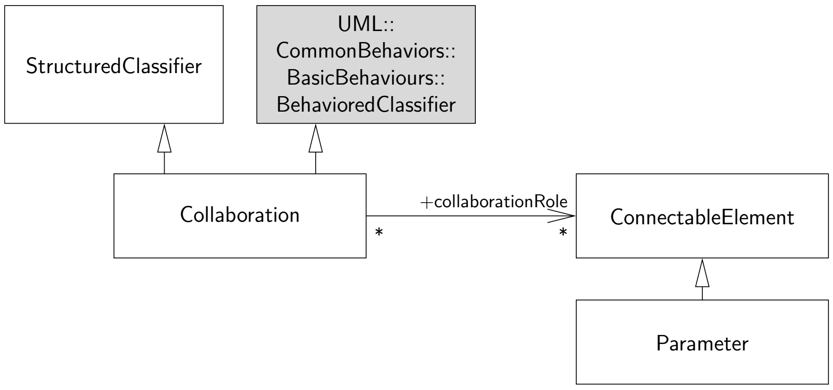


Figure 4.5: Meta-model of UML collaborations (cf. [OMG09b, p. 165]).

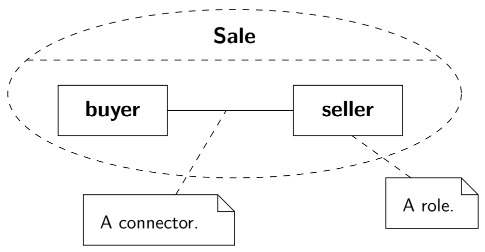
4.2.2 Visualisation

Figure 4.4 shows a composite structure diagram. It describes a simplified version of a car and the decomposition of its power train. Hereby, two rear wheels are connected to the engine through an axle. The axle is plugged to the engine at a dedicated port. Furthermore, gas control is forwarded to an external port of the car. Note, that the provided and required interfaces of ports in a composite structure diagram are given implicitly by the roles of the contained components.

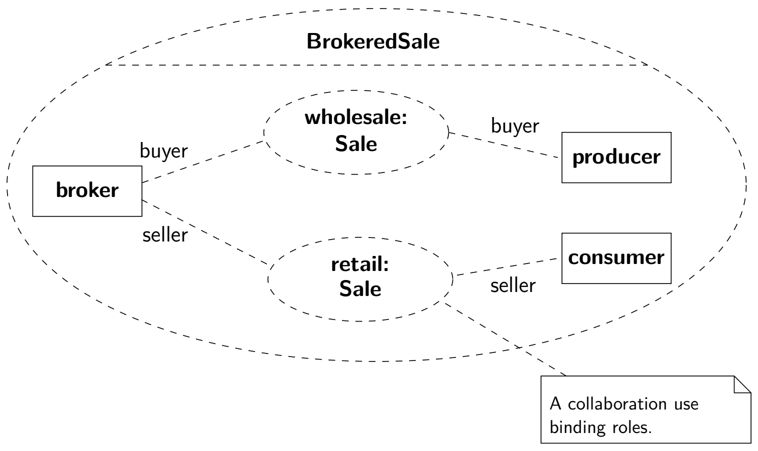
4.3 Collaboration Diagram

4.3.1 Description

A variation of a composite structure diagram is the *collaboration* diagram (cf. Figure 4.5). Collaboration describe the structure of collaborating elements, referred to as *roles*, which collectively achieve some joint task. Role represent elements with specialised functionality, explicitly expressed by typing a role with a classifier. A role is a reduced view on an instance when acting within the collaboration. Thus, the role specifies the minimal set of features a participating instance must have [OMG09b, p. 169]. The connectors between roles describe independent communication links that exist within an instance of a collaboration. As the collaboration itself primarily focuses on the structural composition of elements, it may also contain a behavioural specification in form of, e.g., a sequence diagram. Hence, a collaboration combines both forms of system specification, in structure and behaviour.



(a) Collaboration.



(b) Collaboration use.

Figure 4.6: Example of a UML collaboration (cf. [OMG09b, pp. 172–173]).

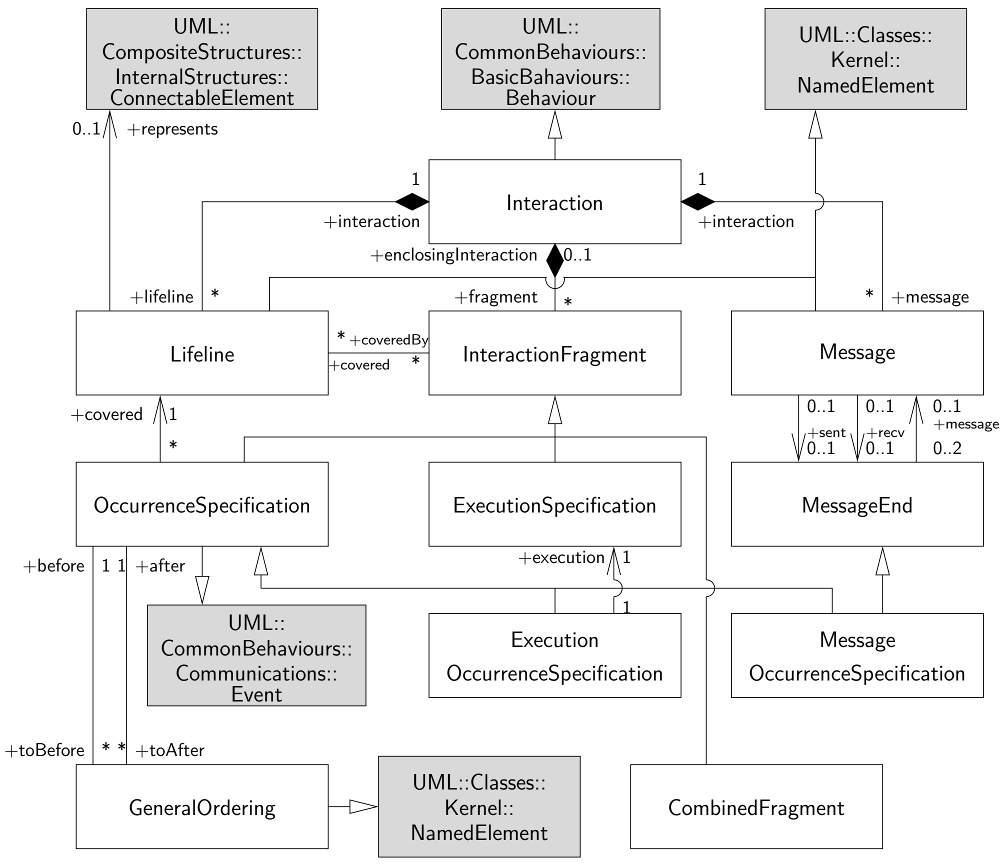


Figure 4.7: Simplified meta-model of UML Interactions (cf. [OMG09b, pp. 462–466]).

4.3.2 Visualisation

Figure 4.6a depicts a simple sale collaboration. It involves two roles, buyer and seller, which act together through one communication link. The other diagram, Figure 4.6b, shows a second collaboration re-using the previous one to model a more complex example, a brokered sale. The sale collaboration is bound two times, as wholesale and as retail, respectively. When binding a collaboration, referred to as *collaboration use*, the roles of the bound collaboration are applied to instances of the binding classifier, e.g., the broker role of the outer collaboration is associated to the buyer role of the bound one (cf. Figure 4.6b).

4.4 Sequence Diagrams

4.4.1 Description

A sequence diagram represents one possibility to model information flow between arbitrary communication partners, may it within one system or in-between systems. This diagram is a variant of a UML interaction diagram. It is used to express fixed execution lines, timely or causally controlled work flows, alternatives or repetitions occurring during communication. Thereby, the focus lies on the exchanged messages. Sequence diagrams are based upon the concept of *interactions*. An “interaction is unit of behaviour that focuses on the observable exchange of information between" at least two parties which communicate with each other [OMG09b, p. 483]. As such, it describes traces of valid or invalid occurrences of communication events, i.e., sending or receiving messages.

Sequence diagrams are used whenever the following aspects of communication shall be emphasised [RQZ07, p. 406]:

- The ordering of messages is important.
- Interactions between the parties are complex.
- State transitions within the communication parties which are caused by messages have only minor relevance.
- The structural binding and how it is established at run-time is irrelevant.
- Details about interaction flow shall be expressed.

Sequence diagrams allow for hiding internal behaviour of the communication parties, regarding them as black boxes. In these cases, only externally observable behaviour is reflected in the diagram. In literature, such sequence diagrams are also referred to as *System Sequence Diagrams* (*SSDs*) (cf. [Lar02, p. 118]).

As depicted in Figure 4.7, an interaction consists of three major parts:

Lifelines. Lifelines are used to identify the roles, i.e., the communicating parties, of the modelled interactions. A lifeline refers to a part of a model which can be connected to other elements by means of establishing a communication with them. For instance, a sequence diagram may represent the message flow for a collaboration. Hence, a lifeline represents a role of that collaboration.

Messages. Messages represent the information flow of an interaction. They model a directed flow of data as the essential part of interactions. Messages are created by a sender and addressed to a receiver, both represented by lifelines.

Messages either model invocation of operations and possible returns, or transmission of signals. While signals always have asynchronous communication semantics, operation invocations are distinguished in being either synchronous and asynchronous. For asynchronous message communication, the sender of the message (or signal, respectively) does not wait for a response to this message from the receiver. Thus, the sender is not blocked by the transmission of the message but immediately continues its own execution. In contrast, for synchronous communication, the sender waits for the receiver to finish message processing and to return a response. While waiting, the sender's execution is paused. Hence, sender and receiver explicitly synchronise their executions by that message exchange.

Typically, messages are only exchanged between owned lifelines of an interaction. However, messages may also be *found*, i.e., the message was initiated by an unknown sender outside the modelled interaction and received by one of the interaction's lifelines. The counterpart to found messages are *lost* messages which are created by a lifeline but their reception is not explicitly modelled by the containing interaction.

Operand	Semantics
Alternative	At least two mutually excluding message traces are modelled.
Option	An optional trace is expressed which is either executed once or never.
Break	A break represents an exceptional trace of an interaction which, when chosen, terminates the containing interaction.
Parallel	Such an interaction fragments defines a number of message sequences which are executed in parallel.
Weak/Strict Sequencing	Such traces either allow slightly reordering of events between multiple lifelines without contradicting the partial order of event occurrences between each pair of lifelines, or they enforce a strict, i.e., total ordering of events as annotated in the model even if the more general interaction semantics would allow other orderings.
Negative	Negative traces explicitly enumerate invalid behaviour.
Critical Region	To model atomic blocks of traces, use a critical region. Such regions can not be interleaved by other event occurrences.
Ignore/Consider	These regions are used to emphasise special traces of an interaction.
Assertion	The modelled traces of an assertion represent the only valid continuations of an interaction. All other continuations result in invalid behaviour.
Loop	The described set of traces is repeated a number of times.

Table 4.1: Interaction operands for combined fragments.

Interaction Fragments. Finally, the third part of interactions are interaction fragments. An interaction fragment is the most general unit of an interaction. It is used as the glue between lifelines and messages. Therefore, UML introduces the notion of occurrence specifications. Such a specification models the occurrence of some event within an interaction. This is either a message event caused by sending or receiving a message, or it is an execution event which is triggered when starting of finishing an operation of a lifeline. The executions themselves are also modelled via interaction fragments.

A third group of interaction fragments are combined fragments. Conceptually, combined fragments represent an interaction by themselves (cf. [OMG09b, pp. 469 and 487]). Combined fragments express a region of an interaction which applies to special rules, determined by an interaction operand. This operand influences the sequence or frequency of messages additionally to the general semantics of interactions. The UML defines a fixed set of operands, listed in Table 4.1.

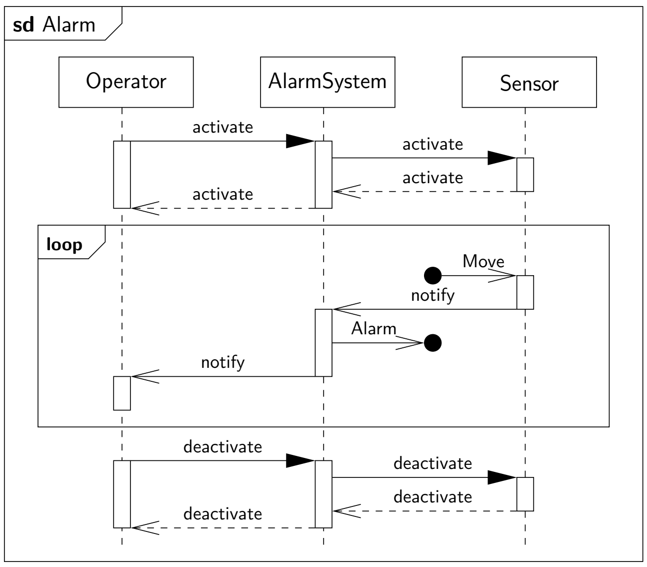


Figure 4.8: Example of a UML Sequence Diagram (cf. [RQZ07, p. 435]).

4.4.2 Visualisation

Figure 4.8 depicts a sample sequence diagram of an alarm system. The hereby modelled interaction identifies three roles as lifelines: the Operator, the AlarmSystem, and the system's Sensor. The operator can activate the system, which in turn activates the sensor. This interaction is done synchronously by invoking the activate cascade. Once being active, the system reacts on external events, i.e., the sensor detects movements in the system's environment. This is modelled via the found message Move. Upon detection, the sensor asynchronously notifies the alarm system which triggers an external alarm, expressed as lost message Alarm. For instance, this alarm would be forwarded to a police station but remains opaque for the alarm system itself. Next, the system propagates the sensor's notification back to the operator. Eventually, as for activation, the operator can synchronously deactivate the whole system.

Note, that the alarm chain is enclosed in a loop fragment. Thus, the alarm may occur several times while the system is active. As the loop fragment is unguarded, it is also a valid trace of the system if no move at all is detected and hence, no alarm is triggered until the system is deactivated.

4.4.3 Relation to Message Sequence Charts

Message Sequence Charts (*MSCs*) provide a language for the specification and description of communication behaviour of system components and their environment in form of message traces [oI01]. They represent an international standard established by the *International Telecommunication Union* (*ITU*) to describe real-time communication within telecommunication switching systems. MSC is a formal language which supports both, a textual and a graphical representation. Additionally, the ITU described the languages formal semantics [oI01] to enable automated analysis by appropriate tools.

One major point of criticism for sequence diagrams in UML 1.0 was the lack of clear semantics and unclear handling of the diagram's elements. Hence, when developing UML 2.0,

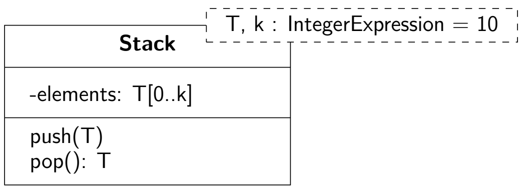


Figure 4.9: A stack as UML template.

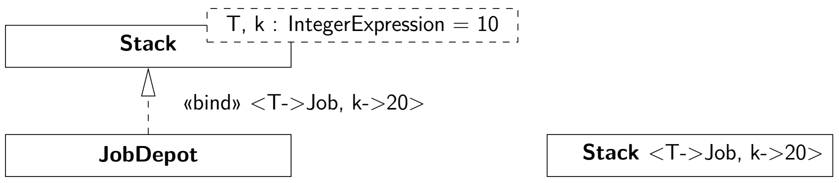
the OMG orients itself on MSCs, adopting various concepts and notations. In consequence, UML 2.0 sequence diagrams and MSCs are closely related to each other. Although the UML still does not define formal semantics for sequence diagrams, they are technically quite similar to MSCs as outlined by Øystein Haugen who is one of the major contributors behind both standards (cf. [Hau05] for an in detail discussion). The bottom line is, that UML sequence diagrams can basically be converted to a corresponding MSC and thus inherit a formal grounding.

4.5 Templates

The template mechanism provides means for parametrisation of UML models. In UML 2.0, there exist three types of elements which can be represented as templates, i.e., classifiers, packages, and operations. Therefore, a template defines a set of template parameters. These parameters fulfil two tasks: First, they add genericity to models in terms of substitutable elements with can be used to configure models when applied to specific contexts. Second, they represent place-holder elements which are used within a template model itself. Thus, a template can reference generic, i.e., intentionally underspecified, aspects of the model. Template parameters can represent classifiers, value specifications, properties, or operations.

A template is visualised like the UML model it parametrises. Its signature is shown as a rectangle superimposing the upper right corner of the diagram and containing the template's parameter declarations. Figure 4.9 depicts a model of a stack. The stack has two template parameters. The type of elements to be stored in the stack remains generic, given by the classifier template parameter T. Additionally, the value specification parameter k is used to configure the maximum number of elements to be hold by the stack at the same time. Note, that template parameters support the specification of default values. Hence, if the number of elements is not explicitly bound upon template instantiation, a default of 10 elements can be stored in the stack.

A concrete model is derived from a template by a template binding. Figure 4.10 shows both versions offered by UML in the context of the stack example. Either, a concrete stack, like the JobDepot is explicitly bound, or the stack is instantiated by an anonymous class (cf. Figure 4.10a and 4.10b, respectively). In either case, the resulting UML model describes a stack which may contain up to 20 elements of type Job. The advantage of the anonymous binding is, that it allows the direct typing of a property using the template itself and thus avoiding the need to define a separate model just to instantiate the template [OMG09b, pp. 634–635].



(a) Explicit binding. (b) Anonymous binding.

Figure 4.10: Binding a UML template.

4.6 Extending the UML

UML 2 allows the tailoring of the language to support special needs of system modelling. For example, given a *Domain Specific Language* (*DSL*), UML can be adopted to support modelling in alignment to such a DSL. By directly supporting its own extensibility, UML provides a flexible way of re-using its constructs within a new context without the need to directly modify the standard itself.

The extension mechanism supports *lightweight* or *heavyweight* extensions to UML. The heavyweight approach allows for modification of the UML meta-model itself. Thus, meta-classes and their relations may be extended, modified or removed. In turn, one will define its own dialect of UML. The advantage of this approach is the ability to specify an own meta-model without further restrictions. However, when modifying the meta-model, one has to ensure, that the resulting meta-model does not contradict preserved parts of UML.

The more common way of adopting the UML to new domains is using the lightweight approach, referred to as *profiling* [OMG09a, p. 177], [OMG09b, p. 653]. By defining a profile, UML meta-classes are referenced by *stereotypes*, which extend their meta-class by additional attributes or constraints. Stereotypes may also derive from other stereotypes. The profiling concept was specified for a number of motivations where a straightforward mechanism for adapting UML seemed to be advantageous [OMG09b, p. 654]:

- Add DSL specific terminology to UML.
- Define customised element syntax for elements previously lacking a concrete notation.
- Adjust or replace notations for existing elements.
- Clarify semantics of semantic variation points within special context.
- Add completely new semantic concepts to UML.
- Add further conditions and stronger restrictions to model elements than already defined by the meta-model.
- Add MDA relevant annotations to model elements, e.g., controlling model transformation processes.

As depicted in Figure 4.11, a profile is a UML package owning stereotype definitions. A stereotype specifies how a UML meta-class is extended by domain specific concepts, e.g., presenting specific terminology or additional properties. Meta-class extension is done through the Extension association, explicitly linking meta-classes and stereotypes. Furthermore, alternative graphical notations may be introduced for stereotypes by decorating them with

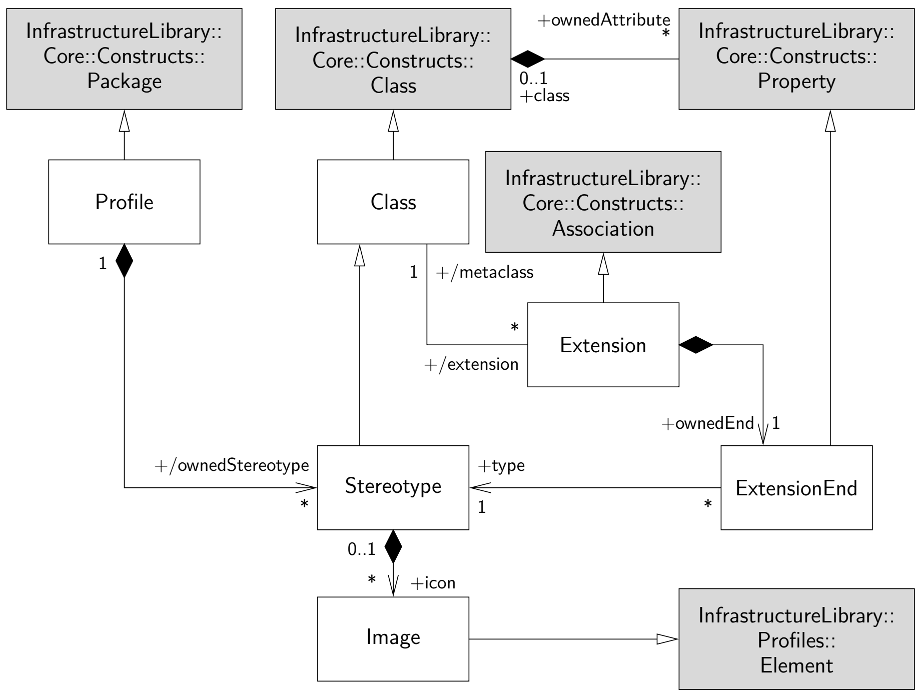


Figure 4.11: Simplified meta-model of UML profiles (cf. [OMG09b, p. 656] and [OMG09a, p. 180]).

Images. As a profile is derived from package, a profile may define domain specific data types and further constraints which must be fulfilled by model elements which are effected by this profile.

Figure 4.12 gives an example of a profile definition and its application. To apply stereotypes of a profile to model elements, the profile is referenced via an «apply» association. Then, stereotypes can be applied to model elements being instances of a compatible meta-class with respect to the stereotype's meta-class. The depicted profile defines a new stereotype «servicecomponent», extending the UML meta-class Component by an additional property id. The Components package employs this stereotype. MyService is an instance of a UML component, tagged with the new stereotype. To set the value for id, one uses a comment annotation.

Stereotypes do not define new meta-classes. Instead, an element tagged by a stereotype is an instance of the stereotype's referenced meta-class enriched with the stereotype's properties and constraints. A stereotype basically extends an existing UML meta-class by a special notion of inheritance. A stereotype may extended its meta-class by additional attributes or it may restrict values of attributes defined by the meta-class. It is important to note, that restrictions applied by a stereotype to a meta-class must not contradict the UML meta-model. For instance, a stereotype can not completely remove an attribute of its meta-class. This simple technique allows for easy tool support of UML language extensions. As the UML meta-model remains untouched by profiling, existing UML tools can be used without modifications to support customized UML extensions which are based on profiles. Even if a tool does not support the handling of profiles itself, a UML model being enriched by profile stereotypes can still be processed by the tool as it is a standard UML compliant model.

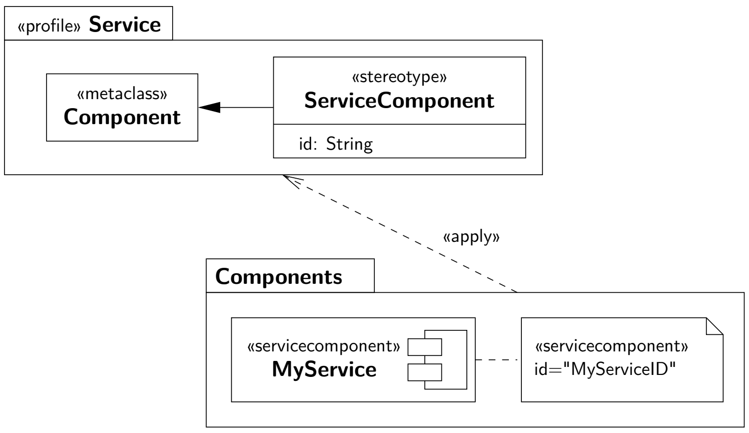

«profile» Service
«metaclass» Component
«stereotype» ServiceComponent
id: String
«apply»
Components
«servicecomponent» MyService
«servicecomponent» id="MyServiceID"

Figure 4.12: UML example.

However, the stereotype's added properties and constraints will not be understood by such tools.

Stereotypes of profiles can be classified by their role with, or purpose for, a UML model as well as by their expressiveness. In literature, there is a differentiation between three major roles of stereotypes (cf. [SK05, SK06]):

Model Transformation/Code Generation. Stereotypes of this category aim to control various aspect of model transformation and code generation. Based on the stereotype, or its properties, a modelling tool may chose from alternative transformation rule sets or different variants of reflecting a model in a concrete programming language.

Virtual Meta-model Extension. Extensional stereotypes are applied whenever the UML is to be adopted for a new modelling domain. Thus, stereotypes can be used to add a new vocabulary to the UML.

Model Simplification. Stereotypes can be used to simplify models. In that sense, a stereotype encapsulates special requirements on modelling elements by hiding modelling constraints behind its own definition. Thus, the model itself is not polluted with repetitive annotations of the same restrictions. A simplification stereotype may also denote varying roles of stereotyped elements in the design. Such a stereotype makes design principles explicit to the model.

Orthogonal to their role within a model, stereotypes are also classified with respect to their influence to a model's semantics, i.e., their expressiveness. Early in the development of UML, Bremer et al. identified four functional categories of stereotypes (cf. [BGJ99]):

Decorative. A decorative stereotype does not add (restrict) any semantics to (of) the UML meta-model. It just changes the syntax of modelling elements.

Descriptive. A descriptive stereotype add symbolic information to the model about the intention or pragmatics of a newly introduced concept. It does not modify the models semantics.

Restrictive. A restrictive stereotype applies additional constraints to the UML meta-model, refining the semantics of UML elements.

Redefining. A redefining stereotype changes the semantics originally applied to a modelling element. As such, a redefining stereotype does not represent a lightweight extension to UML.

Part II

Modelling Interaction-Centric Services

CHAPTER 5

The Modelling Framework

This chapter presents the new service development process. The first part gives an overview of the key activities of the process itself and their necessary design steps. Next, the essential modelling elements behind the process along with their formal definitions and their corresponding representations in UML are described.

5.1 The Interaction-Centric Service Development Process

Based on the concept of ITs, the underlying principle of the proposed service development process is the strict separation between the specification of services and systems – as compositions of services – and the definition and realisation of the interactions occurring within a services. On the one hand, this allows for service specification based on generic interactions rather than being oriented on concrete features of the addressed target platform. On the other hand, generic interaction descriptions are mapped to primitives of such target platforms based on their characteristic interaction semantics and independent of their concrete application within a service.

Consequently, ITs play two different roles. First, they represent an abstract platform [Alm06] for service specification. That is, ITs form an open library of building blocks which can be re-used to describe a service's interactions in a platform independent manner. And second, each IT defines an abstract model which itself needs to be realised on a target platform. Note, that in its first role, an IT is a *platform specific* primitive as part of a virtual *interaction platform* for services. In contrast to this, an IT, as seen in its second role, itself represents a *platform independent* model which is to be realised on a lower-level *target platform*.

Figure 5.1 highlights this separation of concerns, also reflected by distinguishing different stakeholders within the process: the service designer and the system designer, both basing their work on the IT library to specify services and systems, respectively. Both actors are complemented by the IT designer, being tasked with the setup and maintenance of this library through the specification of ITs and appropriate target mappings used to ground service interactions to target platforms.

The stakeholder's concrete tasks are depicted in Figure 5.2 which details the individual

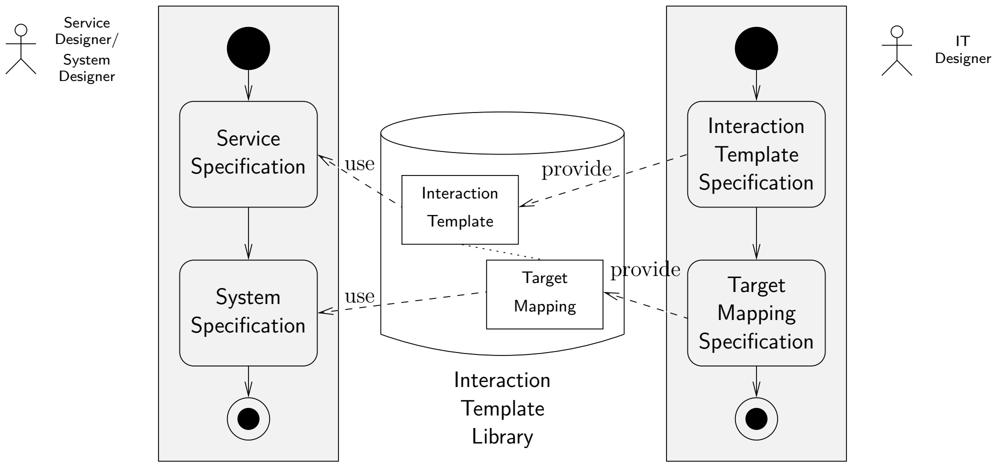

Service Designer/ System Designer
Service Specification
System Specification
use
use
Interaction Template
Target Mapping
provide
provide
Interaction Template Library
Interaction Template Specification
Target Mapping Specification
IT Designer

Figure 5.1: The key activities of the modelling process.

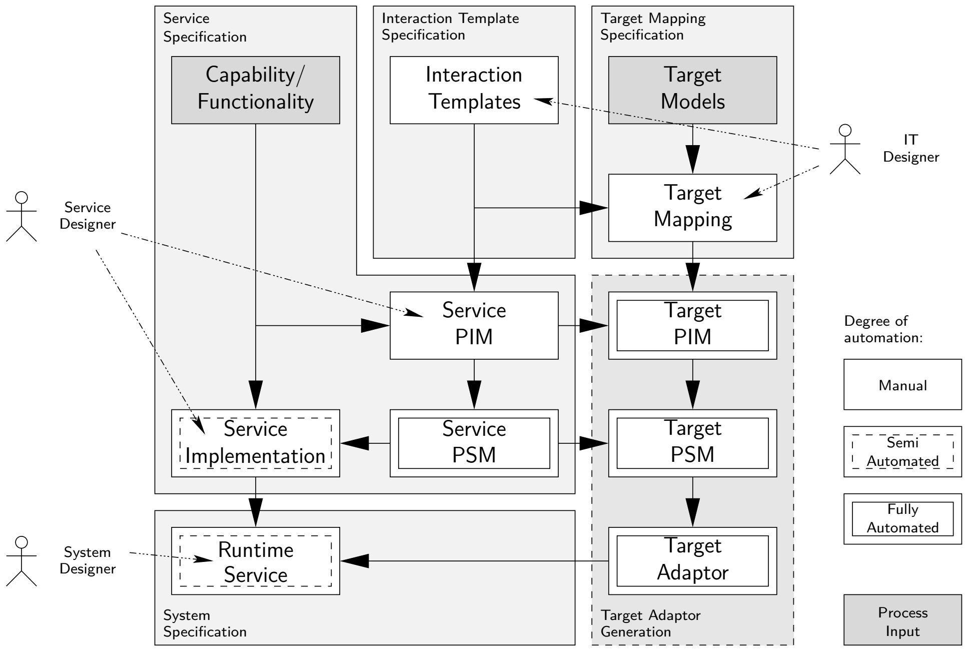


Figure 5.2: Inputs and outputs of the modelling processes.

activities of the development process by giving an overview of their individual inputs and outputs, also visualizing the relationships between the (intermediate) models handled by the proposed process. Each activity is outlined in a separate sub-section, clarifying the entities of Figure 5.2. Additionally to the already introduced activities, the figure shows a fifth one – the target adaptor generation which establish the grounding of a service's concrete interactions to target platforms. This activity is fully automated within the described process and thus is not explicitly assigned to one of the actors.

5.1.1 Service Specification

Services represent collaborative functionality realized through interactions between the service's participants, referred to as *service roles*. The service designer decides how such a particular collaboration shall be realised by identifying this set of interactions. Consequently, the service model represents a composition of interactions between service roles. These interactions are derived from ITs via template instantiation. Hereby, an interaction pattern captured by an IT is applied to the specific context of the modelled service, i.e., by providing concrete messages to be exchanged as part of the interaction and by assigning the interaction's roles to service roles. The resulting set of IT instantiations, forms the service PIM. Throughout the rest of this thesis, we simply use the term *interaction* to refer to an IT instantiation for use in a service.

After the service designer has chosen a programming language to implement the service's roles, the service PSM is automatically derived from the respective service PIM by model transformation. The service PSM represents the service's roles and interactions in terms of the chosen programming language. Based on the service PSM, stub/skeleton code is generated which is used for the actual service implementation, resulting in deployable components. Note, that a complete service implementation consists of a number of such components, one for each service role, as the service itself represents a collaboration of multiple entities.

5.1.2 System Specification

The task of the system designer is to assemble a system specification out of service implementations. Therefore, he selects service components. As these components are based only on the platform independent representations of the services' interactions, they need to be combined with appropriate target adaptors, the interactions' platform bindings, which ground the interactions to primitives of the addressed deployment platforms. In order to establish the services at run-time, a target adaptor per service interaction is necessary.

The relations between the inputs and outcomes of the individual activities concerning service and system design are exemplified in Figure 5.3, explicitly highlighting the combination of service components and target adaptors. Note, that each interaction of a service is individually mapped to a separate target adaptor and thus, possibly, to different target platforms which is illustrated by the ragged lines in the depicted adaptors. Depending on the run-time environment the specified system will be executed on, even multiple adaptors may be chosen for just one interaction. This allows for an interaction to be established via varying technologies at run-time. However, both ends of an interaction must still share at least one common realisation through the same target mapping to enable communication at all.

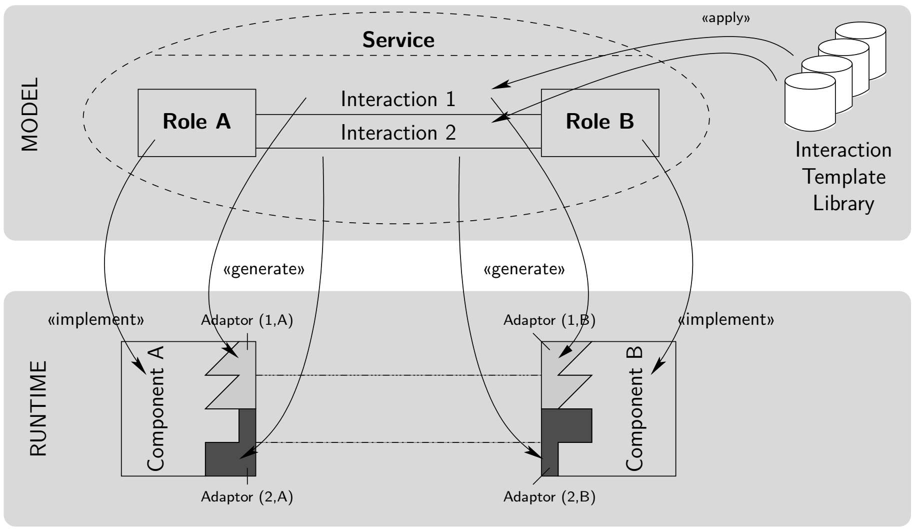


Figure 5.3: The relation between ITs, services and run-time components.

5.1.3 Interaction Template Specification

The IT designer models interaction patterns through ITs. Interaction patterns capture a common interaction scenario between a pair of communicating entities in a generic manner [Bir06], i.e., they do not refer to concrete messages or service contexts. Different ITs describe different interaction patterns. The definition of ITs results in the specification of an abstract platform [ADvSP04] providing high-level primitives for service interaction. In particular, this abstract platform describes an "ideal" platform for service definition by providing platform primitives which directly reflect a service's needs for interaction and thus avoids any necessary wrapper functionality in application logic.

ITs are specified by an IT designer who identifies the underlying semantics of an interaction pattern and creates a model which directly reflects these semantics for use by a service. The captured semantics include the participating entities of an interaction, referred to as *interaction roles*, as well as the generic set of messages being exchange during the establishment of such an interaction but intentionally without identifying concrete messages needed by specific services. Hence, an IT describes both, structural and behavioural features of interaction patterns.

The motivation to specify a new IT may originate from two sources. First, a service designer may be faced with the need for a new interaction pattern while developing a service. As he does not find an appropriate IT already present in the IT library, a new IT has to be defined. Second, a new target platform is adopted for the development process (as a target model) and its own characteristic interaction primitives shall be reflected as ITs, enabling direct usage in service specifications.

5.1.4 Target Mapping Specification

In conjunction with the specification of an IT, the IT designer must also define how the IT is to be realised on a concrete target platform, e.g., via web services or a lower level communication middleware like CORBA. Such a mapping is represented by a target mapping. There may exist more than one such a mapping for a single IT, either to realise the IT on different target platforms or to express mapping alternatives for the same target platform. IT mappings are based on ITs and target models. Target models describe the available interaction semantics of a target platform's primitives, e.g., pure socket communication, message passing or RPC, in terms of modelling concepts, e.g., the UML profile for CORBA [OMG02b]. Based on these primitives, a target mapping is defined by model transformation rules, describing how an instantiation of an IT within a service specification is transformed to represent it on the target platform. For instance, an IT describing the semantics of an RPC may be intuitively mapped to a *Remote Method Invocation* (*RMI*) of a CORBA object. A target mapping may also include directives to generate semantic wrapper code if a target platform does not natively support an IT's semantics, e.g., when realising an RPC directly on top of socket communication.

5.1.5 Target Adaptor Generation

The target adaptor generation sub-process implements a target mapping specification in the context of a concrete interaction within a service. All necessary steps are fully automated. In a first step, the target PIM is created. The target PIM is a derived model which describes a service's interactions in primitives of the target platform. More precisely, the target PIM is the result of a model transformation process which uses the service PIM as a source model. The transformation is controlled by the corresponding ITs' target mappings. The resulting target PIM may be, for instance, a CORBA IDL model which describes an interaction's roles as CORBA objects.

Although the target PIM describes interactions in a target platform specific manner, it remains still quite generic as it is not influenced by implementation details. Similar to the transformation of the service PIM to the service PSM to enrich the model with such knowledge, the target PIM is transformed to the target PSM. Thereby, the gap between the general mapping of IT semantics to the target platform primitives and a service role's concrete implementation is closed. Thus, the target PSM refines the target PIM by adding information about how an interaction's realisation is to be connected to a service role's implementation. This information is retrieved from the service PSM as a second source model next to the target PIM. Following the example of a CORBA IDL mapping from above, the target PSM can be seen as the corresponding Java mapping generated by the IDL compiler.

Eventually, as derivative of the target PSM, the target adaptor is generated. The target adaptor represents the final result of the adaptor generation sub-process. It is an executable component implementing an interaction's role for a target platform. It is linked to the service role's implementation to realise an interaction at run-time – executed on the selected target platform.

Beside the pure mapping of a service's interactions to target platform primitives, both models, the target PIM and the target PSM, may include additional information. The target PIM can contain semantic wrappers as specified by the corresponding IT mapping to close semantic gaps between an IT's semantics and the platform's primitives. Additionally, the target PSM will most likely specify details about implementation "glue code", which converts elements of the service PSM to elements of target PSM, such as type conversions or interface delegates. To illustrate the necessity for such glue code, assume a service interaction is to be realised by CORBA and the service's roles are implemented in Java. This implies two consequences: First, the interaction's messages must be translated between target plat-

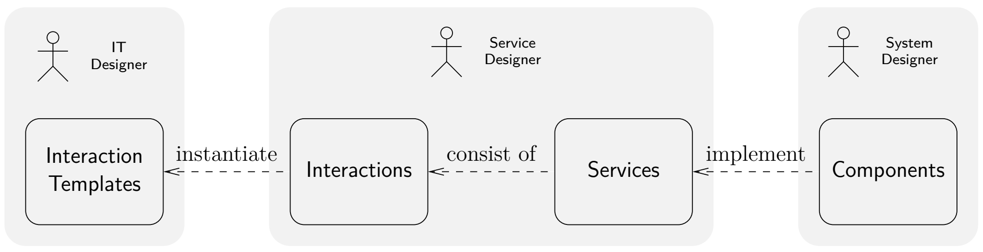


Figure 5.4: The relationships between the first class modelling entities.

form independent Java representations and their CORBA representatives. And second, the interaction roles' CORBA interface definitions must be adapted to the service roles' Java signatures.

Note, that depending on the selected target platform, the target PSM may only subtly differ from the target PIM if the target platform already closely matches the service role's implementation language, e.g., when implementing the service roles with Java and realising the interactions with Java RMI.

5.2 The First Class Modelling Entities of the Development Process

The following sections describe the primary modelling entities of the presented service development process, depicted in Figure 5.4. They represent the first class entities to be created by the IT designer, i.e., the ITs as such, the entities under responsibility of the service designer, i.e., interactions and services, and eventually the entities specified by the system designer, i.e., components.

Note, that the target mappings are, within the context of this thesis, assumed to be directly represented by model transformation rules executed by a transformation engine and are not reproduced as formal models. Additionally, a formal model for complete system specifications is intentionally omitted as questions such as component selection, distribution, deployment, execution, and endpoint reference handling are out of scope of this thesis.

5.2.1 Interaction Templates

Formal Definition

ITs model interaction patterns. As such, they "describe the core structure of [an interaction] solution at a level high enough to generalize to many specific situations [and which] can be tailored to fit" [vdBC01, p. 265] in the very specific context of a concrete interaction scenario. More precisely, ITs document similarities of interactions between two entities in terms of communication interfaces and semantics.

An IT is built-up on the basis of *actions* which cluster an interaction pattern's underlying principles of message exchange. "An action is a named element that is the fundamental unit of executable functionality. The execution of an action represents some transformation or processing" [OMG09b, p. 236]. Hence, actions describe logically indivisible, causally related exchanges of messages, which either provide *input* to or represent *output* from the action's processing. An action has a direction. It is initiated by one entity participating in the interaction, referred to as *source*, and addressed to another entity of the interaction, referred to as *destination.* Input messages of an action are sent from the source to the destination and thus trigger the execution of the action. Output messages are returned from the destination to the source as outcome of the action.

Let T be the non-empty set of message type definitions. Then, an action is defined as:

Definition 2 (Action)**.** *An action is defined as a triple* $(id, \mathcal{T}_{in}, \mathcal{T}_{out})$*, where*

id *is the unique identifier for this action.*

$\mathcal{T}_{in}$ *is an n-tuple* $(i_1, \ldots, i_n) \in T^n$ *typing the input messages sent by this action.*

$\mathcal{T}_{out}$ *is an m-tuple* $(o_1, \ldots, o_m) \in T^m$ *typing the output message returned by this action.*

An action $(id, (i_1, \ldots, i_n), (o_1, \ldots, o_m))$ *is written as:*

$$id(i_1, \ldots, i_n) : o_1, \ldots, o_m$$

For an action a, one writes $id(a)$, $\mathcal{T}_{in}(a)$, and $\mathcal{T}_{out}(a)$ to denote the identifier, the n-tuple of input message types, and the m-tuple of output message types of a respectively.

The presented definition of actions focuses on the exchanged message types and leaves the action's processing behaviour intentionally undefined. The action's functionality is generalised to a pure causal relation of messages, i.e., the output messages are caused by the input messages. Additionally, a message is also generalised to its type rather than being expressed by its concrete content.

Actions are divided into *formal* and *actual* ones. Formal actions are used within ITs as template parameters. In this context, a formal action describes the occurrence of an action within an interaction pattern. In contrast, an actual action represents a concrete action as part of an interaction which is derived from an IT. Note, that the message types defined for a formal action act as place-holders to represent the pure existing of an input or output message. Concrete message types are defined by actual actions upon replacement of formal actions.

Based on formal actions, an IT allows the specification of an interaction pattern describing communication between two entities, referred to as *interaction roles.* Let R_{it} be the non-empty set of interaction roles, and let A_f and A_a be non-empty and pair-wise disjoint sets of actions. Then, an IT is defined as:

Definition 3 (Interaction Template)**.** *An* Interaction Template *(*IT*) is defined as a 5-tuple* $(\mathcal{R}_{it}, \mathcal{A}_f, D, c, B_{it})$*, where*

$\mathcal{R}_{it}$ *is a set* $\{r_1, r_2\} \subseteq R_{it}$ *of two interaction roles representing the communicating entities of the captured interaction pattern.*

$\mathcal{A}_f \subseteq A_f$ *is a set of* formal actions *of the IT.*

$D \subseteq \mathcal{R}_{it} \times \mathcal{A}_f$ *is the binary "is-destination-of" relation. D identifies interaction roles of an IT as action destinations within this IT. Every formal action is assigned to exactly one interaction role, i.e., D is right-total and left-unique (cf. Appendix B).*

We define $\overline{D}$ as complement of D identifying interaction roles as action sources. $\overline{D}$ is given by:

$$\{(r,f)|f \in \mathcal{A}_f \land r \in \mathcal{R}_{it} \land (r,f) \notin D\}$$

$\overline{D}$ is also right-total and left-unique.

Clearly, an interaction role is either an action's destination or source, but not both at the same time, i.e., $D \cap \overline{D} = \emptyset$

c *is a predicate $c : \mathcal{A}_f \times A_a \rightarrow \{true, false\}$ being true if a given actual action is a valid replacement for a formal action of this IT upon IT instantiation.*

B_{it} *is a behaviour specification defining temporal and causal relations of message exchanges captured by the IT's actions in form of an MSC.*

For an IT i, one writes $\mathcal{R}_{it}(i)$, $\mathcal{A}_f(i)$, $D(i)$, $\overline{D}(i)$, $c(i)$, and $B_{it}(i)$ to denote the set of interaction roles, the set of formal actions, the destination and source mappings, the action replacement constraints, and the behaviour of i respectively.

Primarily, an IT identifies a set of formal actions, a set of roles, and the relations between both. The mappings D and $\overline{D}$ guarantee that every action has a well defined and unique destination and source, guaranteed by right-totalness and left-uniqueness. A role may be in relation to more than one formal action either as a destination or as a source. An IT's roles are symbolic identifiers for entities collaborating through the interaction pattern captured by the IT. The IT's formal actions $\mathcal{A}_f$ provide means for applying the captured pattern to a concrete interaction. They represent place holders to be substituted by actual actions dependent on the interaction's context. Such actual actions describe concrete types of messages being transmitted when establishing an interaction at run-time, e.g., an actual action specifies concrete input and output messages for an RPC. As one can see by this example, a single action of an IT may cause a number of actual message transmissions, e.g., first sending the input message and next receiving the return message.

The expansion of actions to individual message transmissions is defined by the IT's behaviour specification B_{it}. B_{it} defines how an action's message exchanges in-between interaction roles are interleaved by causal and temporal dependencies as well as synchronisation semantics, like blocking and non-blocking method invocations. Internal details about the behaviour of individual interaction roles are not covered by B_{it}. Roles are regarded as black boxes. The focus lies on externally observable communication behaviour. For instance, in the case of a *Request/Response* (*R/R*) pattern, the reception of a request message will cause a new execution for the callee which lasts until the callee sent back the response message. Hereby, the observable behaviour of callee is represented by a sole execution specification, framed the by request/response message pair. However, an implementation of the callee may have complex internal behaviour, e.g., by spanning separate worker threads to detach each incoming request. Note, that in the context of the presented thesis, the focus for B_{it} specification lies on capturing the general behaviour of an interaction role sufficiently detailed to characterize the modelled interaction pattern. A more sophisticated, formal specification of behaviour is not in the scope of this thesis.

An IT may restrict the replacement of formal actions by actual ones by enforcing additional constraints on actions. For instance, the mandatory absence of output messages for an action if the IT's semantics do not support a response channel to propagate a result back to the initiator of the interaction. Such constraints are enforced by the c predicate, which provides a validation test for formal to actual action replacements.

UML Representation

ITs are visualised by UML collaboration template diagrams. Hereby, interaction roles $\mathcal{R}_{it}$ of an IT directly map to collaboration roles. An IT's formal actions $\mathcal{A}_f$ become UML operations, supporting input and output messages via operation parameters. These operations are defined as template parameters of the afore mentioned collaboration template as a direct consequence of the formal actions' place-holder semantics.

The IT's "is-destination-of" relation D is reflected by typing the collaboration roles with UML interfaces. These interfaces group all operations which represent actions addressed to this role. As these operations are template parameters of the collaboration diagram itself, they can not be used directly to define the operations for a collaboration's role interface. Instead, these role interfaces are templates themselves, also mirroring the corresponding actions for a role as operation template parameters. Next, when using these interfaces as a collaboration role's type, the appropriate operation template parameters from the surrounding collaboration are delegated to bind the interface templates. This approach allows for modelling an action addressed to an interaction role as *provided* operation of the associated collaboration role. If an interaction role is not the destination of one of the IT's actions, the respective interface of its collaboration role remains empty and can be omitted in the collaboration diagram.

Note, that the described approach of using collaboration templates stays in contrast to the informal notion of "collaboration templates" used in the UML standard. Hereby, the template parameters of the collaboration are interface template parameters which directly represent the types of the collaboration roles [OMG09b, p. 634]. The drawback of this solution w.r.t. to our work is that arbitrary interfaces can be used to type roles. Thus, it can not be guaranteed that a role's type is compatible with respect to an interaction role's provided operations.

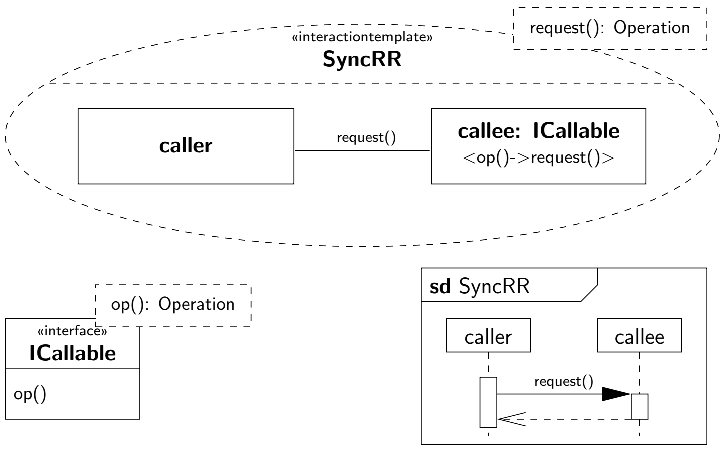
If the substitution of one of the formal actions by an actual action is constrained within the IT, as specified by the predicate c, the constraint is reflected by an *Object Constraint Language* (*OCL*) expression. This expression is annotated as invariant to the corresponding operation template parameter of the collaboration template. Thus, the substitution constraints become automatically evaluable by a modelling tool.

Finally, an IT's behaviour is modelled by a UML sequence diagram as used for SSDs (cf. Section 4.4.1 on page 39). The diagram is directly referenced by the collaboration diagram as its owned behaviour specification. The lifelines of the sequence diagram correspond to the collaboration roles, and thus to the interaction roles. An action's message exchanges are expressed by message occurrences within the sequence diagram. To draw a stronger relation between the dynamic and structural specification of an IT, every single action in the IT results in a connector between the collaboration roles. This connector is then referenced by the action's messages in the sequence diagram. This way, we express distinct communication channels for each action.

Figure 5.5 depicts an example of an IT specification in both variants, the formal 5-tuple (a) and as UML model (b). Note, that the notation $\uparrow$*SyncRR* provides a short-cut for the behaviour specification B_{it} of an IT. Is is used to refer to the corresponding MSC as formal

$$
\begin{aligned}
IT_{SyncRR} = (\mathcal{R}_{it} &= \{caller, callee\},\\
\mathcal{A}_f &= \{request()\},\\
D &= \{(callee, request())\},\\
c(f,a) &= true,\\
B_{it} &= \upharpoonright SyncRR \quad)
\end{aligned}
$$

(a) Formal specification.



(b) UML representation.

Figure 5.5: Example of an Interaction Template specification.

specification of an IT's semantics. As introduced in Section 4.4, a well defined subset of UML can be used to describe such MSCs by UML sequence diagrams. For clarity, only this UML diagram is shown when documenting an IT. Thus, the diagram is substituted by $\upharpoonright SyncRR$ or similar expressions for an IT's formal specification.

The *SyncRR* IT captures a synchronous R/R interaction pattern (cf. to Chapter 7 for a detailed discussion about this IT). The pattern describes a blocking method invocation, initiated by a caller and addressed to a callee interaction role. The invocation action of the IT is modelled as single formal operation template parameter request() which is not affected by any constraints during IT instantiation. Hence, the c predicate is simply defined as constant *true* and is not reflected as OCL invariant to the operation template parameter. As the caller interaction role is never addressed by an action, it is simply typed by an empty interface which is intentionally not displayed in the diagram. In contrast, the callee interaction role is typed by the intermediary ICallable interface which provides the sole operation op(), configurable through a template parameter. This operation is bound to the request() operation of the enclosing collaboration template. Finally, the behaviour specification is given by the sequence diagram SyncRR, stating that request() is a synchronous message, preventing execution continuation by the caller after being sent until the reply message is received.

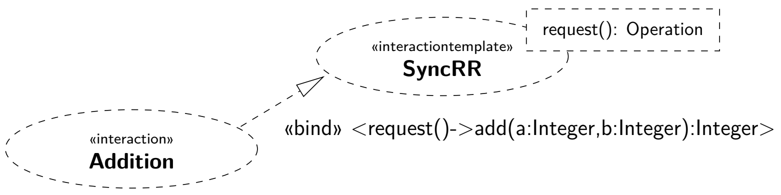
5.2.2 Interactions

Formal Definition

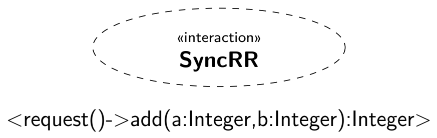
As previously indicated, a concrete interaction describes an IT instantiation, i.e., providing an actual action for all formal actions of the IT such that the validation test succeeds. Let

$$\begin{aligned} I_{Addition} = (IT &= IT_{SyncRR}, \\ \mathcal{A} &= \{(request(), add(Integer, Integer) : Integer)\}) \end{aligned}$$

(a) Formal specification.



(b) Explicit IT binding.



(c) Anonymous IT binding.

Figure 5.6: Example of an interaction specification (IT instantiation).

I be the non-empty set of ITs. The formal definition of an interaction is then given as:

Definition 4 (Interaction)**.** *An interaction, or IT instantiation, is defined as a pair $(IT, \mathcal{A})$, where*

$IT \in I$ *is an IT from which the interaction is derived.*

$\mathcal{A} \subseteq \mathcal{A}_f(IT) \times A_a$ *is the binary "is-replaced-by" relation. $\mathcal{A}$ identifies exactly one actual action as replacement for each of the formal actions of the referenced IT as part of its instantiation. $\mathcal{A}$ is left-total, left-unique, and right-unique (cf. Appendix B).*

Given $\mathcal{A}$, we can extract the set of actual actions of an interaction $\mathcal{A}_a$:

$$\mathcal{A}_a = \{a | a \in A_a \wedge f \in \mathcal{A}_f(IT) \wedge (f, a) \in \mathcal{A}\}$$

Let c_{it} be the action replacement test predicate of IT, i.e., $c_{it} = c(IT)$. An interaction is valid*, if all its action replacements satisfy the IT's replacement constraints, i.e.,*

$$\forall (f, a) \in \mathcal{A} \quad c_{it}(f, a) = true$$

By being left-total, the "is-replaced-by" relation $\mathcal{A}$ ensures, that every formal action is bound to an actual action. Furthermore by also being left- and right-unique, $\mathcal{A}$ replaces a formal action by an actual one in a one-to-one relation – every formal action is replaced by exactly one actual action and every actual action is used to replace only one formal action. This results in a complete instantiation of an IT by unique actual actions.

For an interaction i, one writes $IT(i)$, $\mathcal{A}(i)$, and $\mathcal{A}_a(i)$ to denote the IT of the instantiation, the action replacement definitions as well as the set of actual actions defined by i, respectively.

UML Representation

Based on an IT's collaboration template we can derive a concrete interaction via a UML template binding. Consequently, interactions are represented by plain UML collaborations.

For instance, we can define an *Addition* interaction based on the SyncRR IT by providing the actual action add which sums up two integer values, provided as input messages, and returns the result as output message, see Figure 5.6a. In UML this interaction can be defined either explicit by a newly named collaboration (Figure 5.6b) or anonymous (Figure 5.6c).

Interaction Platform Bindings

The realisation of interactions on a target platform is given by an appropriate target adaptor. The adaptor will provide the interaction's platform binding, i.e., the concrete protocol and data representation to be used when the interaction is established at run-time. Although the interaction itself is still platform independent, the platform binding is not. The bindings are defined by the target mapping specifications which complement IT models and hence are inherited by interaction models.

Which concrete target mappings will be used depends on the final system configuration, i.e., available middleware, and will be expressed by model annotations to the individual interactions[1] to steer the development process' model transformations.

5.2.3 Services

Formal Definition

Services represent an abstract view on collaborative behaviour. As such, a service identifies two features: First, the entities which participate in the collaboration, referred to as *service roles*. And second, the interactions in-between these roles to establish a particular functionality. A service defines at least two roles but is not limited to this number of collaborating entities. Thus, the traditional bi-partition of roles in provider and consumer is weakened in the presented thesis, e.g., a provider is not limited to just provide functionality for a service but also may use functionality of other roles from within the same service. The service's roles communicate with each other to establish the modelled collaboration. This communication occurs via interactions, thereby an interaction always connects two of the service's roles. Hence, a service role is constructed as the aggregation of interaction roles of interactions it participates in within a service.

Let R_{svc} be the non-empty set of service roles pair-wise disjoint from interaction roles, i.e., $R_{it} \cap R_{svc} = \emptyset$. Let I_{inst} be the non-empty set of interactions (IT instantiations), and $M \subseteq R_{it} \times R_{svc}$ a binary relation between interaction and service roles. Then, the formal notion of services is given by the following definition:

Definition 5 (Service). *A service is defined as a triple* $(\mathcal{R}_{svc}, \mathcal{I}_{inst}, \mathcal{M})$*, where*

$\mathcal{R}_{svc}$ $\subseteq R_{svc}$ *is the set of* roles *defining symbolic roles of entities participating in the service. A service identifies at least two such roles, i.e.,* $|\mathcal{R}_{svc}| \geq 2$*.*

$\mathcal{I}_{inst}$ *is an n-tuple* $(i_1, \ldots, i_n) \in I_{inst}^n$ *of pair-wise disjoint interactions (IT instantiations) possibly occurring within the service.*

$\mathcal{M}$ *is an n-tuple* $(m_1, \ldots, m_n) \in M^n$ *with* $n = |\mathcal{I}_{inst}|$*. An element* $m_i \subseteq \mathcal{R}_{it}(IT(i_i)) \times \mathcal{R}_{svc}$ *is the binary "is-mapped-to" relation mapping the interaction roles of the i-th interaction*

[1]Represented by the realisation attribute of the InteractionUse stereotype (see Section 6.2.6)

in $\mathcal{I}_{inst}$ to different roles of this service. Every such m_i is left-total, left-unique, and right-unique (cf. Appendix B).

Furthermore, for every service role exists at least one mapped interaction role from one of the interactions, i.e.,

$$\underset{r_{svc}\in\mathcal{R}_{svc}}{\forall}\ \underset{r_{it}\in R_{it}}{\exists}\ \underset{1\leq i\leq|M|}{\exists} : (r_{it}, r_{svc}) \in m_i$$

To ensure, that every interaction within a service is used to interconnect two different service roles, the “is-mapped-to” relations m_i have some specific characteristics. Every such relation guarantees, that both interaction roles are mapped to service roles, i.e., m_i is left-total. Furthermore, an interaction role is only mapped once, to exactly one service, i.e., m_i is right-unique. Thus, not two different service roles can play the same interaction role within one interaction. Next, by being also left-unique, m_i ensures that not both interaction roles of an interaction map to the same service role. Otherwise, if an interaction’s roles are mapped to only one service role, this interaction would be hidden within a service role’s implementation as it would not be observable as part of the service’s collaborative behaviour. Finally, every service role must play at least one role in an interaction within the service. This prevents the definition of functional isolated service roles not communicating with other service roles within the collaboration. Clearly, a service role may have more than just one interaction role mapped to it. Then, it will be involved in multiple interactions within the service, subsuming interaction roles of different interactions.

For a service s, one writes $\mathcal{R}_{svc}(s)$, $\mathcal{I}_{inst}(s)$, and $\mathcal{M}(s)$ to denote the set of service roles, the referenced interactions, and the interaction role to service role mappings of s respectively.

UML Representation

Like ITs and interactions, services are visualised as UML collaboration diagrams. A service’s roles are modelled as collaboration roles. These roles are connected through UML collaboration uses which reference the service’s interactions and assign interaction roles to service roles. Figure 5.7 depicts an example of a service definition. The Calculation service identifies two service roles, the operator and the calculator. The two roles are connected by a pair of interactions based on the SyncRR IT as introduced in Section 5.2.1. The operator can request two types of calculations from the calculator, the addition of two integer numbers or their multiplication respectively. Therefore, the IT’s formal request() action is substituted by add(a:Integer,b:Integer):Integer in the first case and multiply(a:Integer,b:Integer):Integer in the second case upon IT instantiation. Furthermore, both interactions’ roles are bound in a similar manner: the interactions’ caller role to the operator service role and callee to calculator.

Note, that a UML collaboration use has explicit names identifying the application of a UML collaboration, i.e., multiplication and addition in the given example. These names are provided for UML compliance of the diagrams and are not a mandatory part of the service specification itself.

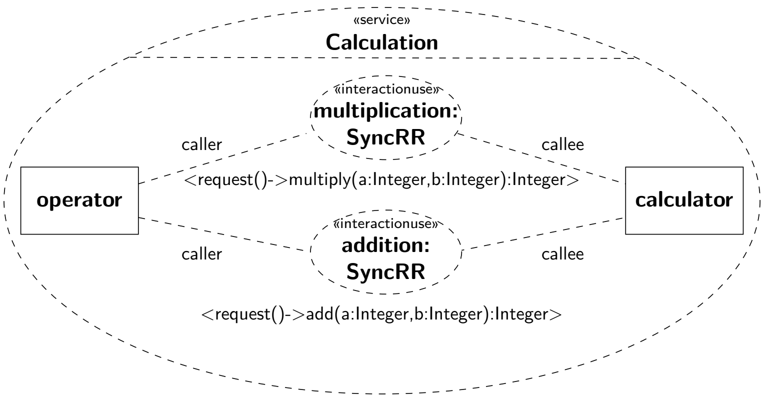
5.2.4 Components

Formal Definition

Components form the conceptual element to realise a service role. Thus, components represent partial implementations of services. Components are part of a service-oriented system

$$
\begin{aligned}
Svc_{Calculation} = (\mathcal{R}_{svc} &= \{operator, calculator\}, \\
\mathcal{I}_{inst} &= ((IT = IT_{SyncRR}, \\
&\quad \mathcal{A} = \{(request(), \\
&\qquad multiply(Integer, Integer) : Integer)\}), \\
&\quad (IT = IT_{SyncRR}, \\
&\quad \mathcal{A} = \{(request(), \\
&\qquad add(Integer, Integer) : Integer)\})), \\
\mathcal{M} &= (\{(caller, operator), (callee, calculator)\}, \\
&\quad \{(caller, operator), (callee, calculator)\}))
\end{aligned}
$$

(a) Formal specification.



(b) UML representation.

Figure 5.7: Example of a service specification.

specification. Hereby, a system designer splits the modelled system into distinct parts, i.e., components, which are then interconnected by services. A component is linked to a service role through a *port*, representing a dedicated point of interaction between the component and its environment.

Let S be the non-empty set of services. Then a port is defined as:

Definition 6 (Ports)**.** *A port is defined as pair* $(\mathcal{S}, \mathcal{R})$*, where*

$\mathcal{S} \in S$ *is a service linked to the port.*

$\mathcal{R} \in \mathcal{R}_{svc}$ *is a service role defined by* $\mathcal{S}$ *to be fulfilled by a component through this port.*

For a port p, one writes $\mathcal{S}(p)$ and $\mathcal{R}(p)$ to denote the service and service role linked to p, respectively.

Let P be the non-empty set of ports and $\mathcal{P}(P)$ be the power set of P. Then a component is defined as:

Definition 7 (Component)**.** *A component is defined as a pair* $(id, Ports)$*, where*

id *is the unique identifier of the component.*

$Ports \in \mathcal{P}(P)$ *is the set of ports of the component.*

For a component c, one writes $id(c)$ and $Ports(c)$ to denote the identifier and set of ports of c, respectively.

A component participates in a number of service interactions based on the service role bindings from its ports. For each of these ports, we derive a type definition by evaluating the port's role binding. The type of a port is composed by *provided* and *required* interfaces. Through a provided interface of a port a component offers functionality to other components participating in the service referenced by the port. Hence, a provided interface is given by the set of actions of which the port's service role is a destination of. Hereby, a service role is an action's destination if the action is addressed to an interaction role which is bound to this service role. The contrary applies for a port's required interface which identifies the actions initiated by interaction roles bound to the port's service role.

To define provided and required interfaces formally, we define a couple of helper functions. $destOf$ and $sourceOf$ are functions $I \times R_{it} \rightarrow \mathcal{P}(A_f)$ retrieving the sets of formal actions of an IT for which a given interaction role is a destination or source of, respectively. Let $i \in I$ be an IT and $r \in \mathcal{R}_{it}(i)$ be an interaction role thereof. Then, $destOf$ and $sourceOf$ are defined as:

$$destOf(i, r) = \{f | f \in \mathcal{A}_f(i) \wedge (f, r) \in D(i)\}$$
$$sourceOf(i, r) = \{f | f \in \mathcal{A}_f(i) \wedge (f, r) \in \overline{D}(i)\}$$

Both functions are also defined for interactions, i.e.,IT instantiations, to return actual actions instead of formal ones for a given interaction role, i.e., $I_{inst} \times R_{it} \rightarrow \mathcal{P}(A_a)$. Let $i \in I_{inst}$ be an interaction and $r \in \mathcal{R}_{it}(IT(i))$ be one of its interaction roles inherited from the underlying IT. Then, using the previous definitions of $destOf$ and $sourceOf$ the extended functions are given by:

$$\begin{aligned}
destOf(i,r) =& \{a | a \in \mathcal{A}_a(i) \land f \in \mathcal{A}_f(IT(i)) \land \\
& (f,a) \in \mathcal{A}(i) \land f \in destOf(IT(i),r)\} \\
sourceOf(i,r) =& \{a | a \in \mathcal{A}_a(i) \land f \in \mathcal{A}_f(IT(i)) \land \\
& (f,a) \in \mathcal{A}(i) \land f \in sourceOf(IT(i),r)\}
\end{aligned}$$

Another function $role_{it} : S \times R_{svc} \times \mathbb{N} \rightarrow \mathcal{P}(R_{it})$ is defined to retrieve the interaction role a service role is mapped to for a specific interaction within a service. Let $s \in S$ be a service specification, $r \in \mathcal{R}_{svc}(s)$ a service role of s, i_j the j-th interaction specification in $\mathcal{I}_{inst}(s)$ $(1 \leq j \leq |\mathcal{I}_{inst}(s)|)$, and m_j the corresponding service role mapping relation in $\mathcal{M}(s)$. Then, $role_{it}$ is defined as:

$$role_{it}(s,r,j) = \{r_{it} | r_{it} \in \mathcal{R}_{it}(IT(i_j)) \land (r_{it}, r) \in m_j\}$$

The function $role_{it}$ will either return the empty set or a set of just one element. If the empty set is returned, the service role r is not in any relation to an interaction role of the j-th interaction in service s, i.e., it does not participate in this specific interaction of the service. Otherwise, it is guaranteed that at most a single element of R_{it} is returned, because every "is-mapped-to" relation m_j as element of $\mathcal{M}(s)$ relates both interaction roles of an IT to different service roles (cf. Definition 5). Hence, for a given role mapping relation m_j there exists at most one such pair (r_{it}, r) for a service role r.

Based on these helper functions, we now define provided and required interfaces for ports. Let $p \in P$ be a port specification and $\mathcal{I}_{inst}(\mathcal{S}(p)) = (i_1, \ldots, i_n)$ the tuple of interactions defined by the service the port is linked to.

Definition 8 (Provided Interfaces of a Port)**.** *The port's* provided interfaces *are defined by the n-tuple:*

$$provided(p) = (p_1, \ldots, p_n) \in (\mathcal{P}(A_a))^n \quad n = |\mathcal{I}_{inst}(\mathcal{S}(p))|$$

An element p_j $(1 \leq j \leq n)$ of this tuple is the set of actual actions of the j-th interaction in $\mathcal{I}_{inst}(\mathcal{S}(p))$, denoted as i_j, where the port's service role is a destination of, i.e.,

$$p_j = \bigcup_{\forall r \in role_{it}(\mathcal{S}(p), \mathcal{R}(p), j)} destOf(i_j, r)$$

Definition 9 (Required Interfaces of a Port)**.** *The port's* required interfaces *are defined by the n-tuple:*

$$required(p) = (r_1, \ldots, r_n) \in (\mathcal{P}(A_a))^n \quad n = |\mathcal{I}_{inst}(\mathcal{S}(p))|$$

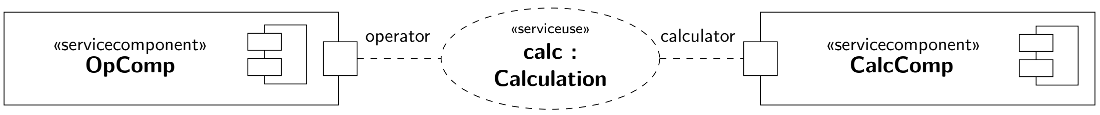
An element r_j $(1 \leq j \leq n)$ of this tuple is the set of actual actions of the j-th interaction in $\mathcal{I}_{inst}(\mathcal{S}(p))$, denoted as i_j, where the port's service role is a source of, i.e.,

$$r_j = \bigcup_{\forall r \in role_{it}(\mathcal{S}(p), \mathcal{R}(p), j)} sourceOf(i_j, r)$$

$$\begin{aligned} Comp_1 &= (id &&= OpComp,\\ & \quad Ports &&= \{(\mathcal{S} = \uparrow Svc_{Calculation},\\ &&& \quad \mathcal{R} = operator)\}) \end{aligned}$$

$$\begin{aligned} Comp_2 &= (id &&= CalcComp,\\ & \quad Ports &&= \{(\mathcal{S} = \uparrow Svc_{Calculation},\\ &&& \quad \mathcal{R} = calculator)\}) \end{aligned}$$

(a) Formal specification.



(b) UML representation.

$$\begin{aligned} provided(Port_{OpComp}) &= (\{\}, \{\})\\ required(Port_{OpComp}) &= (\{multiply(Integer, Integer) : Integer\},\\ & \quad \{add(Integer, Integer) : Integer\}) \end{aligned}$$

$$\begin{aligned} provided(Port_{CalcComp}) &= (\{multiply(Integer, Integer) : Integer\},\\ & \quad \{add(Integer, Integer) : Integer\})\\ required(Port_{CalcComp}) &= (\{\}, \{\}) \end{aligned}$$

(c) Provided/Required interfaces.

Figure 5.8: Example of components implementing the Calculation service.

A port has a number of required and provided interfaces according to the number of interaction roles bound to the port's service role. Note, that these interfaces may be empty, i.e., when a service role does not provide or require any actions for or from a specific interaction. According to Definition 8 and Definition 9, empty required and provided interfaces are possible for a specific combination of a port's service role and an interaction role of the enclosing service's interactions. Empty interfaces occur, when either a port's service role does not participate in an interaction, i.e., $role_{it}$ returns the empty set, or the port's service role does either not provide or require any actions within the specific interaction.

UML Representation

Components and their ports are model by UML component diagrams. Both elements have a direct one-to-one mapping to the appropriate UML elements, i.e., UML components and UML ports, respectively. Services connect components via UML collaboration uses. Hereby, a service is linked to a UML component by a UML port. The UML collaboration use's role assignment expresses the binding of service roles to ports.

Figure 5.8 continues the examples of the previous sections. It shows how two components are defined to establish the Calculation service. On the left, the OpComp component which is linked to the operator service role through an anonymous port. On the right, the calculator service role is fulfilled by the CalcComp component, also through an anonymous port.

A port's provided interfaces can be determined from the UML diagrams. First, we follow a port's service role binding in the Calculation service, e.g., calculator. Next, based on the service specification, two interaction roles are bound to this service role, i.e., caller in the addition as well as in the multiplication interaction, both based on the SyncRR IT. Knowing the interaction roles and the associated IT, we extract their type definitions from the IT's collaboration template diagram. These types represent a port's provided interfaces. To determine a ports required interfaces, we select the opposite interaction role's type definition within an interaction a port's service role participates in. Note, that the type definitions as specified by the IT are UML interface templates which must be properly instantiated with an interaction's actual actions. The whole process of determining a port's interfaces from the formal as well as from the UML models is described in more detail in Section 9.1 as part of our case study.

5.3 Summary

This chapter described the proposed, interaction-centric service development process and modelling entities. The first part presented the process' stakeholders and their key activities: the service designer to specify services, the system designer to compose systems, and the IT designer to build up a library out of interaction template specifications and their corresponding target mappings. Following in the second part, the first class modelling entities of the development process were formally defined along with their representations in UML.

CHAPTER 6

A UML Profile for Service Modelling

This chapter presents the *UML Profile for Interaction-centric Services* (*UP4IS*), a UML 2 profile providing the necessary UML primitives to describe services based on ITs as introduced in Chapter 5. The first part of this chapter describes the "virtual" meta-model for the UP4IS profile. This meta-model exists only as a conceptual basis to illustrate the relations of the profile's individual stereotypes. Thus, this meta-model is not directly reflected in UML nor is it a modification of UML's own meta-model. Instead, it visualises the elements of the formal model introduced in Chapter 5 and their relations to each other as UML compatible structures.

In the second part of the chapter, the individual elements of this meta-model are mapped to concrete UML meta-classes which support the required properties of each element best. This mapping process results in the precise definition of stereotypes for the UP4IS profile.

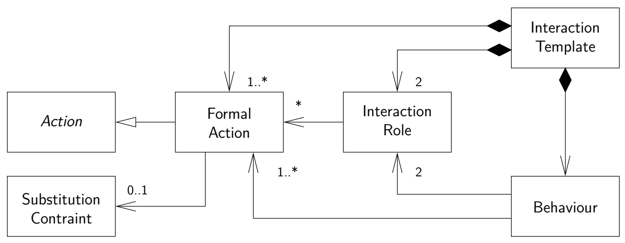
As result, the UP4IS profile provides means to model services based on ITs with standard UML diagrams which are, at the same time, well-formed service models with respect to the presented formal model for interaction-centric services. Due to additional model constraints on the UML meta-model which are defined by the UP4IS stereotypes, the diagrams compliance to the formal model is guaranteed.

6.1 The UP4IS Meta-Model

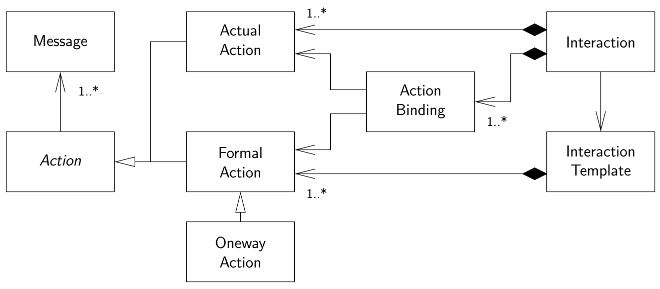
This section presents the meta-model which is provided by the UP4IS profile. The elements and relations within this meta-model reflect the formal model for services based on ITs as introduced in Chapter 5. Figure 6.1 depicts the individual elements of the UP4IS meta-model. Its elements are discussed individually with respect to their relation to concepts of the formal model.

6.1.1 Actions

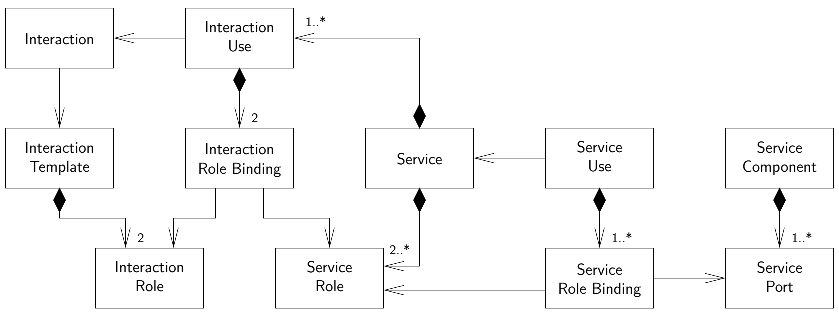
Actions represent atomic blocks of communication between two entities. Formally, as given by Definition 2, an action identifies a set of messages, which either represent an action's



(a) Interaction Templates.



(b) Interactions.



(c) Services and Components.

Figure 6.1: The UP4IS Meta-Model.

input or output. This association is directly reflected in the UP4IS meta-model through an abstract action (cf. Figure 6.1b).

Within the formal model, two types of actions are distinguished: First, formal actions are used for IT specifications. Therefore, formal actions represent placeholder actions and provide means to model causal relations between individual action occurrences in interaction patterns. Second, actual actions are used when deriving concrete interactions from ITs. Hence, actual actions are substitutes for formal actions upon template instantiation. In the UP4IS meta-model, this differentiation between both types of actions is preserved, using abstract action as a common super-type.

6.1.2 Interaction Templates and Interactions

ITs capture interaction patterns between a pair of communication entities. Thus, they represent aggregations of individual actions. As stated in Definition 3, ITs are formally defined by 5-tuples $(\mathcal{R}_{it}, \mathcal{A}_f, D, c, B_{it})$. The translation of this concept is illustrated in Figure 6.1a. An IT's interaction roles, given by $\mathcal{R}_{it}$, are identified by the IT's aggregation relation to such roles. The second aggregation relation, to formal actions, models the IT's set of formal actions as defined by $\mathcal{A}_f$. Each such formal action is referenced by an interaction role. This association reflects the "is-destination-of" relation D of the IT. Additionally, action constraints which may effect formal to actual action substitutions, as expressed by predicate c, are modelled as substitution constraints. Finally, an IT's behaviour specification is also present in the meta-model. Such a behaviour references the IT's interaction roles and formal actions to describe the order between individual action occurrences.

Instantiating an IT leads to the definition of an interaction. According to Definition 4, an interaction is the pair $(IT, \mathcal{A})$. Thereby the former element identifies the instantiated IT, the latter, $\mathcal{A}$, is the relation which provides actual actions as substitutes for each of the formal actions of the IT. This relation is modelled by separate action bindings, one for each substitution of a formal by an actual action. Thereby, the set of actual actions is explicitly defined within an interaction (cf. Figure 6.1b).

6.1.3 Services, Ports and Components

Services describe interactions between at least two communicating entities. Formally, as introduced in Definition 5, a service is defined as triple $(\mathcal{R}_{svc}, \mathcal{I}_{inst}, \mathcal{M})$. The service roles, i.e., $\mathcal{R}_{svc}$, are modelled by an aggregation relation for a service. Formally, these roles are connected by interactions thereby the interaction's roles are assigned to service roles, given by $\mathcal{I}_{inst}$ and $\mathcal{M}$, respectively. These sets are modelled by interaction uses, which identify the applied interactions, and respective interaction role bindings, which establish the role mappings (cf. Figure 6.1c).

Services are then used to model communication between service components. As expressed by Definition 7, components are aggregations of service ports. According to Definition 6, a port, given by the pair $(\mathcal{S}, \mathcal{R})$, identifies a service $\mathcal{S}$ and a role $\mathcal{R}$ of the service which is realised by this port. This link between a port and a service is reflected in the UP4IS meta-model by a service use and a service role binding, comparable to interaction uses and interaction role bindings.

6.2 The UP4IS Stereotypes

This section presents the stereotypes defined for the UP4IS profile. These stereotypes reflect individual elements of the UP4IS meta-model as standard UML meta-classes. Following the classification in [SK05], the presented stereotypes are *restrictive*, *code generation* stereotypes (cf. Section 4.6). This means, the stereotypes are used to slightly modify the semantics of the original meta-classes with respect to possible relations to other meta-classes or values of their attributes. Additionally, they are *transformational*, being used to control later model transformation processes, e.g., to derive transport adaptor realisations. Note, that not all elements of the UP4IS meta-model are explicitly mapped to stereotypes as their semantics are subsumed by standard UML concepts and thus, stereotypes would only have a *decorative* character. The affected concepts are message, which is not represented as a first class modelling entity but solely as part of actions; interaction and service role binding aggregations which are subsumed by the stereotypes for interaction and service use; the action binding aggregation which also becomes part of the interaction definition; and finally a formal action's substitution constraint which is directly modelled as OCL invariants to the formal action.

The UP4IS stereotypes are presented in alphabetical order. Each of the stereotypes is described based on the following pattern:

Description. The description of the stereotype itself and from which UML meta-class it is derived.

Attributes. The stereotype's attributes, including their formal definition and description. These attributes effectively represent the meta-class extension by the stereotype. In alignment with the UML superstructure document [OMG09b], the prefix "/" is used for derived/read-only attributes of meta-classes.

Constraints. The constraints on the UML meta-model introduced by the stereotype. These constraints reflect the stereotype's semantic modifications to the UML meta-model. Beside their description, the constraints are formalised as OCL expressions [OMG07b] in conformance with the UML superstructure [OMG09b]. Please refer to Appendix C (p. 153) for the respective OCL listings.

Notation. The notation of an element within UML diagrams being marked by the specific stereotype.

6.2.1 Action

Description

An action is a UML *operation* (from the UML Templates package) and defines a common base type for actual and formal actions used for interaction specification. This stereotype is not intended for direct use in UML models.

Attributes

None.

Constraints

1. An action has public visibility as an action provides means for external communication of a component.
2. An action's parameters have the direction in, out, or return.

Notation

The notation for an action is a UML operation with stereotype «action».

6.2.2 ActualAction

Description

An actual action is a UML *operation* (from the UML Templates package) and defines an actual action used to derive a concrete interaction, i.e., an IT instantiation. It is a specialisation of an abstract action and represents an action used to substitute formal actions upon template instantiation.

Attributes

None.

Constraints

No additional constraints with respect to action.

Notation

The notation for an actual action is a UML operation with stereotype «actualaction». When displayed upon anonymous template instantiation the stereotype may be omitted.

6.2.3 FormalAction

Description

A formal action is a UML *operation* (from the UML Templates package) and defines a formal action as parameter of an IT. It is a specialisation of an abstract action and represents a placeholder element to be substituted upon template instantiation. A formal action is used, to associate actions with interaction roles, i.e., specifying source and destination relationships, and to define the order of action occurrences within an IT.

A formal action may define additional invariants in form of an OCL expression to further constrain the characteristics of actual actions which will substitute this formal action, e.g., the absence of return messages for one way actions. To determine the substitutes for a given formal action in the context of interaction/service specifications, we define the OCL query `getActualActions()` (cf. Listing 6.1). The query returns all actual actions, i.e., UML operations, replacing an operation stereotyped as formal action via template instantiation.

```
context UP4IS::FormalAction
def : getActualActions() : Set(Operation) =
        TemplateParameterSubstitution.allInstances()
        ->select(formal = self.base_Operation)
        ->collect(actual)->flatten()->asSet()
```

Listing 6.1: OCL expression to determine a formal action's actual substitutions.

Attributes

None.

Constraints

1. A formal action defines no parameters or exceptions.
2. A formal action does not redefine another operation.
3. A formal action does not define pre-, post-, and body- conditions.

Notation

The notation for a formal action is a UML operation with stereotype «formalaction». When displayed as an IT's template parameter the stereotype may be omitted.

6.2.4 Interaction

Description

An interaction is a UML *collaboration* (from the UML Collaborations package and w.r.t. *classifier* from the UML Templates package) and defines a concrete interaction within a service. An interaction is derived from an IT by template instantiation. Therefore, the interaction provides actual actions for all formal actions of the referenced IT.

Attributes

None.

Constraints

1. An interaction is defined by binding an IT.
2. For each formal action of the referenced IT exists an actual substitute. This constraint guarantees left-totalness for an interactions "is-replaced-by" relation $\mathcal{A}$ (cf. Definition 4).
3. An actual action binds exactly one formal action. This constraints guarantees left-uniqueness for an interactions "is-replaced-by" relation $\mathcal{A}$ (cf. Definition 4).
4. A formal action is bound by exactly one actual action. This constraints guarantees right-uniqueness for an interactions "is-replaced-by" relation $\mathcal{A}$ (cf. Definition 4).

5. An interaction is used exactly once for an interaction use as it is uniquely defined for a concrete service.
6. Actual actions of an interaction are compatible to formal actions of the underlying IT (inherited standard constraint of a UML template parameter substitution [OMG09b, p. 630]). This constraint enforces an IT's substitution predicate c (cf. Definition 3).

Notation

The notation for an interaction is a UML collaboration with stereotype «interaction», representing a template binding. Note that an interaction may not be shown directly in a UML diagram but be presented only implicitly as an anonymous template binding within an interaction use.

6.2.5 InteractionTemplate

Description

An interaction template is a UML *collaboration* (from the UML Collaborations package) template (as defined by *classifier* from the UML Templates package) and defines an IT as introduced in Section 5.2.1. Thus, the interaction template describes an interaction pattern between two interaction roles based on formal actions. The underlying communication pattern is given by an attached behaviour specification.

Attributes

None.

Constraints

1. An interaction template has at least one template parameter.
2. All template parameters represent formal actions.
3. An interaction template contains exactly two interaction roles having different types.
4. At least one of the interaction roles is explicitly typed. If an interaction role is typed, its type is an interface. Note, that the first part of this constraint is enforced by the previous constraint as not both role types can be undefined at the same time.
5. An interaction role's interface references only formal actions of the nesting interaction template as its operations.
6. A formal action is associated to exactly one interaction role via a role's interface. This constraint ensures the binding of each formal action to an interaction role as its destination, as defined by an IT's "is-destination-of" relation D (cf. Definition 3).
7. All connectors within an interaction template connect all interaction roles and no other elements. There exists at least one such connector.
8. A formal action is reflected by a dedicated connector having the same name. There exist no other connectors.

9. An interaction template has a behaviour specification in terms of an associated UML sequence diagram whose lifelines correspond to the template's collaboration roles and the template's formal actions are reflected by synchronous and asynchronous messages, transmitted via the collaboration's connectors.

Notation

The notation for an interaction template is a UML collaboration template with stereotype «interactiontemplate».

6.2.6 InteractionUse

Description

An interaction use is a UML *collaboration use* (from the UML Collaborations packages) and expresses the usage of an interaction within a service. The referenced interaction is mapped to one or more target technologies realising the interaction semantics of the underlying IT. These mappings lead to the generation of target adaptors to be used by the service's role implementations to realise communication within the service at run-time.

Attributes

- realisation : String[1..*]
  Identifies the target technologies the interaction use's referenced interaction is realised on.

- configuration : String[1..*]
  Used as a generic container to configure mapping alternatives for a specific interaction within a service. Possible values depend on the referenced IT and shall be documented along with the ITs themselves.

Constraints

1. An interaction use references an interaction.

2. An interaction use specifies at least one target technology.

3. Every interaction role of the referenced interaction is bound to exactly one service role (inherited standard constraint of a UML collaboration use [OMG09b, p. 171]).

4. Within an interaction use, an interaction role of the referenced interaction is bound to a service role by a dedicated role binding. Additionally, both interaction roles are bound to different service roles.

Notation

The notation for an interaction use is a UML collaboration use with stereotype «interactionuse».

6.2.7 OnewayAction

Description

A oneway action is a UML operation (from the UML Templates package) representing a template parameter for ITs. It is a specialisation of a formal action representing a placeholder which can only be substitute by actual actions not defining any output messages. Such actions are widely used for asynchronous communication. Thus, this stereotype provides means for simplification of IT models, as such a substitution constraint does not need to be defined explicitly when using this stereotype.

Attributes

None.

Constraints

1. A oneway action can only be substituted by actual actions defining only input messages or no messages at all. For an actual action a the constraint is similar to testing for $\mathcal{T}_{out}(a) = \emptyset$.

Notation

The notation for a oneway action is a UML operation with stereotype «oneway». In contrast to formal actions, an action defined as a oneway action must be explicitly stereotyped in UML diagrams. Otherwise, the action is assumed to represent an ordinary formal action.

6.2.8 Service

Description

A service is a UML *collaboration* (from the UML Collaborations package) and defines a service as introduced in Section 5.2.3. A service identifies at least two service roles which are exclusively interconnected by interaction uses.

Attributes

None.

Constraints

1. At least two service roles are defined for a service.
2. Only interaction uses are used as role connectors within a service.
3. Interaction uses within a service connect only service roles of the same service and bind all interaction roles (inherited standard constraint of a UML collaboration use [OMG09b, p. 171]).
4. Every service role is connected to at least one interaction use within its enclosing service.

```
context UP4IS::ServicePort
def : getServiceRole() : ConnectableElement =
            CollaborationUse.allInstances()
            ->select(not extension_ServiceUse.
              oclIsUndefined())
            ->collect(roleBinding)
            ->select(client = self.base_Port)
            ->first().supplier
```

Listing 6.2: OCL expression to determine the service role a service port is linked to.

Notation

The notation for a service is a UML collaboration with stereotype «service».

6.2.9 ServiceComponent

Description

A service component is a UML *component* (from the UML Components package) and represents the implementation of service roles. A service component represents a service role through a service port.

Attributes

None.

Constraints

1. A service component defines at least one service port.

Notation

The notation for a service component is a UML component with stereotype «servicecomponent».

6.2.10 ServicePort

Description

A service port is a UML *port* (from the UML Ports package) and relates a service component to a service role which is implemented by the component. A service port is bound to a service role. The OCL helper operation `getServiceRole()` returns the service role a port is connected to (cf. Listing 6.2).

Attributes

- /provided : Interface[0..*]
  Derived value. Subsets provided from the UML port and references the interfaces being

provided by the nesting component by means of a service role binding to this port (cf. Definition 8 on page 64).

```
context UP4IS::ServicePort
derive: let serviceRole : ConnectableElement =
              self.getServiceRole()
        in serviceRole.clientDependency
           ->collect(supplier.type)
```

- /required : Interface[0..*]
  Derived value. Subsets required from the UML port and references the interfaces being required by the nesting component by means of a service role binding to this port (cf. Definition 9 on page 64).

```
context UP4IS::ServicePort
derive: let serviceRole : ConnectableElement =
              self.getServiceRole(),
            interactions : Collection(Collaboration) =
              Collaboration.allInstances()->select(
              not extension_Interaction.oclIsUndefined())
        in serviceRole.clientDependency->collect(supplier)
           ->collect(r | interactions
             ->select(i | i.collaborationRole->includes(r))
             ->collect(i | i.collaborationRole->reject(r))
             ->flatten()
             ->collect(role | role.type))
           ->asSet()
```

Constraints

1. A service port is bound to exactly one service role via a service use.

Notation

The notation of a service port is a UML port with stereotype «serviceport». When displayed as a service component's port and graphically bound via a service use, the stereotype may be omitted in the UML diagram.

6.2.11 ServiceUse

Description

A service use is a UML *collaboration use* (from the UML Collaborations package) and expresses the usage of a service within a system specification. A service use connects service components' ports to roles of a service. A service may only be partially bound within a system specification. That means, not all service roles must be realised within a system. However, a service is only established, if missing service roles are contributed by other systems.

Attributes

None.

Constraints

1. A service use references a service.

Notation

The notation for a service use is a UML collaboration use with stereotype «serviceuse».

6.3 Summary

This chapter presented a mapping of the modelling entities as introduced by this thesis' novel service modelling and development process to UML. The mapping is realised as a light-weight UML extension in form of a UML profile, referred to as the *UML Profile for Interaction-centric Services* (*UP4IS*). The profile allows for direct use of UML compliant modelling and transformation tools when implementing the proposed development process as applied and demonstrated for the case study in Part III of the presented thesis.

Part III

Case Study

Introduction to the Case-Study

The following chapters will demonstrate the novel development process for interaction-centric services. It will be shown, how the proposed process facilitates service development by supporting a strong separation between platform independent service interaction specification and the realisations thereof on target platforms. Therefore, the following key features are exemplified:

1. Support for the creation of an IT library through capturing of a representative set of interaction patterns. It is to be highlighted, that the specification of ITs and this the creation of this library is an integral part of the development process. The process itself does not demand for a "reference" library serving as standard interaction library. Instead, the library may vary depending on the application context of the process.

2. Support the platform independent specification of services based on ITs, i.e., interaction patterns for service interaction are selected based on a service's need rather than being predetermined by a potentially limited set of interaction primitives of some target platform.

3. Support the implementation of service roles through components by automated model transformation and analysis combined with code generation.

4. Support service deployment by automated generation of service interaction groundings to selected target platforms in form of target adaptors.

Chapters 7 and 8 address the creation of the IT library and service specification, respectively. Service specification is demonstrated based on a simple video recording system. Chapter 9 especially focuses on the process' applied model transformations in the context of the modelled sample services.

CHAPTER 7

Defining an Interaction Template Library

As introduced in Section 5.1, the core element of the proposed design process is a library of ITs. All activities described as part of service design orient themselves to this library. In consequence, it is essential for our case study, to define such a library which can then be used for service specification and which stimulates the proposed model transformation processes.

7.1 Motivating the IT Library

In the following sections, an prototypical IT library is presented. The described ITs are based on the interaction pattern catalogue first published in [Fai98]. This catalogue gives a rough classification of various interaction patterns between a pair of communicating entities. The patterns selected for the following library form a representative selection of the referenced work. The presented library is not complete in terms of covering every imaginable interaction pattern. Instead, as stated in Section 5.1, the library is intended to be open and extensible whenever one identifies the need to describe a new pattern. The presented selection of interaction patterns was chosen to demonstrate applicability of our approach for the following use cases:

- Model primitive interaction patterns. Primitive interaction patterns, like pure message passing (notification) is widely found in literature as the sole explicit modelling primitive for service interaction. By integrating it into the sample library, compatibility to such approaches is effectively shown.
- Model complex interaction patterns. One of the major goals of the proposed modelling process is to hide complex interaction by only one modelling primitive. Thus, service designers are freed of the need to handle complicated communication behaviour explicitly in application logic. Examples for such complex patterns include monitorable *Request/Response* (*R/R*) patterns.
- Model structurally equal but semantic different patterns as ITs. For instance, this is shown by presenting the R/R pattern in various incarnations which can only be distinguished through their behaviour specification.

7.1.1 Documenting ITs

ITs are documented with a consistent format, leading to a uniform structure of the library. Each IT's documentation is divided into the following parts (cf. [GHJV95]):

Name. The IT's name should reflect the essence of the captured pattern to become part of the vocabulary shared between involved participants within the service development process.

Context. The context states the motivation for the described IT. It answers the questions what kind of interaction problems can be solved by this pattern and what is the rationale behind it.

Forces. Forces express further requirements to interaction participants or exchanged messages within the context of the pattern.

Roles. Each IT identifies two roles whose interaction is generalised by the pattern.

Formal Actions. An IT defines a set of formal actions which represent the communication flow between the interaction roles. Each such action is bound to one of the roles, i.e., the role is the action's destination.

Formal Specification. The formal specification of the IT, including symbolic representations of the roles and the formal actions, the "is-destination-of" relation between both, the action substitution constraints, and finally a referenced behaviour specification. See Definition 3 in Section 5.2.1 for more details about these elements.

UML Model. The visualisation as UML model of the IT.

Target Mappings. The set of target mapping specification for the IT, used to realise a derived service interaction on a concrete target platform. These mappings control the model transformation process to generate the associated target adaptors.

Due to the close relation of UML sequence diagrams to the formal grounding of message sequence charts (cf. Section 4.4), the explicit formal behaviour specification for the presented ITs are omitted. Instead, refer to the UML sequence diagram presented as part of the IT's UML model, indicated by the link notation "$\uparrow$" in the formal specification of an IT, e.g., $\uparrow SyncRR$.

7.1.2 Specifying Target Mappings

To exemplify target mappings for ITs, three different target platforms were chosen. In particular, the library provides mappings for Java, Java RMI, and CORBA. The mapping for Java will illustrate how service interaction can be realised within a closed run-time environment avoiding external communication at all. Thus, service components are directly attached to each other, i.e., a component's required interfaces are directly resolved from the "opposite" component. By also handling Java RMI, interaction in distributed environments becomes possible. Although, each service component is still implemented with the same programming language, communication can be realised beyond process or host boundaries. Finally, the

Formal Action	Java/Java RMI Mapping	CORBA Mapping
synchronous	native Java method	IDL operation
asynchronous	detached Java method	IDL oneway operation

Table 7.1: Generic Target Mapping for synchronous/asynchronous actions.

mapping to CORBA allows for heterogeneous implementations of service components, intermixing programming languages. CORBA represents a fully featured networking middleware with its own communication stack, addressing schemes and message marshalling.

The target models for the Java and Java RMI mappings are based on OMG's meta-model and UML profile for Java published as OMG standard specification in [OMG04]. Similarly, for CORBA the corresponding UML profile specification of the OMG is chosen, defined by [OMG02b].[1] The general idea of the target mappings, independent from the concrete target platform, is that an interaction role becomes a dedicated interface declaration expressed in the target technology, i.e., either a Java or a CORBA interface, respectively. Similar to their representation as UML operations, actions become Java methods or CORBA operations of the associated interfaces. An action's concrete mapping relies on its formal semantics. Thereby, two types of actions are distinguished: synchronous and asynchronous ones. Synchronous actions block execution for the initiating party and asynchronous actions allow continuation of execution right after initiation. In case of CORBA, synchronous actions are intuitively mapped to ordinary CORBA operations. In contrast, asynchronous actions result in *oneway* operations, providing best-effort semantics for an asynchronous communication as required by that kind of actions [OMG08a]. Both Java mappings also natively support synchronous actions. But in contrast to CORBA, Java does not provides language means for asynchronous method invocation. To compensate this gap, the Java mappings include wrapper functionality to explicitly detach the invocation of such methods[2]. Table 7.1 summarises the possible mappings depending on action semantics. Individual target mappings of ITs within the following library refer to this table as shortcut.

7.2 The IT Library

7.2.1 Synchronous Request/Response

Context

A process P_1 sends a message, the request, to a process P_2. P_1 must wait until P_2 finished processing. After P_2 finished message processing, it returns a reply message, the response, back to P_1 and P_1 resumes execution.

Forces

Both processes implicitly synchronise their executions to each other as P_1 is blocked while the request is processed by P_2. P_1 fully trusts P_2 to finish processing and return control

[1]Note, that originally, both UML profile specification are defined for UML 1.3. Hence, they were slightly modified to match the UML 2.0 profile specification.

[2]For technical details of these mappings confer to Appendix D.1.

within acceptable time. If P_2 never returns, P_1 may be blocked forever.

Roles

- *Caller*. Caller triggers the interaction by initially sending the request message.
- *Callee*. Callee processes the request and returns the response message.

Formal Actions

- *request*(). The action's input messages represent the request, its output messages the response, respectively. The action is directed to the callee role.

  The action allows substitution by arbitrary actual actions, i.e.:

  $$c(request(), a) = true$$

Formal Specification

$$\begin{array}{rcll} IT_{SyncRR} & = & (\mathcal{R}_{it} & = & \{caller, callee\}, \\ & & \mathcal{A}_f & = & \{request()\}, \\ & & D & = & \{(callee, request())\}, \\ & & c(f, a) & = & true, \\ & & B_{it} & = & \upharpoonright SyncRR) \end{array}$$

Figure 7.1: Formal specification of the Synchronous R/R IT.

UML Model

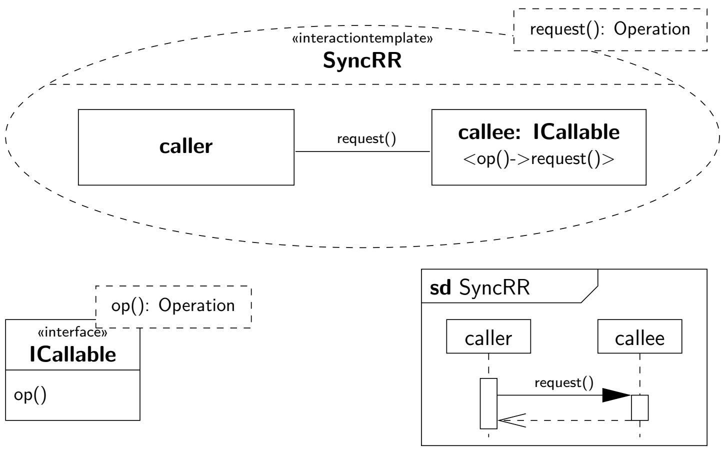


Figure 7.2: UML model of the Synchronous R/R IT.

Target Mappings

Formal Action	Java/Java RMI Mapping	CORBA Mapping
$request()$	native Java method	IDL operation

Table 7.2: Target mapping for the Synchronous R/R IT.

7.2.2 Notification

Context

A process P_1 sends a message, the notification, to be processed by another process P_2 and immediately continues execution. P_1 will gain no direct feedback about the outcome of P_2's processing of the message.

Forces

P_1 is not interested in the result of P_2's message processing. Thus, P_1 continues execution right after sending the message to P_2. In particular, both processes advance completely independent from each other. P_2 may already be busy with some execution when P_1's message arrives. This implies, that P_2 must implement at least some pseudo-parallel behaviour, enabling parallel handling of multiple notification messages.

Roles

- $Notifier$. Notifier sends the notification.
- $Notifyee$. Notifyee processes the notification message.

Formal Actions

- $notify()$. The action's input messages represent the notification to be communicated between notifier and notifyee. It is directed to the notifyee role.

  The formal action identifies only messages being sent from notifier to notifyee, omitting a communication channel in the opposite direction. Thus, only actions not specifying any output messages represent valid substitutes, i.e.:

$$c(notify(), a) = \begin{cases} true & \text{if } \mathcal{T}_{out}(a) = \emptyset \\ false & \text{otherwise} \end{cases}$$

Formal Specification

$$
\begin{array}{lllll}
IT_{Notification} & = & (\mathcal{R}_{it} & = & \{notifier, notifyee\}, \\
 & & \mathcal{A}_f & = & \{notify()\}, \\
 & & D & = & \{(notifyee, notify())\}, \\
 & & c(f,a) & = & \begin{cases} true & \text{if } \mathcal{T}_{out}(a) = \emptyset \\ false & \text{otherwise} \end{cases}, \\
 & & B_{it} & = & \uparrow Notification)
\end{array}
$$

Figure 7.3: Formal specification of the Notification IT.

UML Model

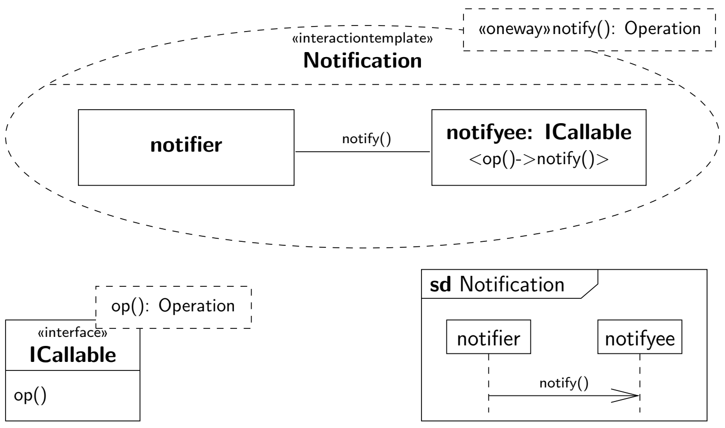


Figure 7.4: UML model of the Notification IT.

Target Mappings

Formal Action	Java/Java RMI Mapping	CORBA Mapping
$notify()$	detached Java method	IDL oneway operation

Table 7.3: Target mapping for the Notification IT.

7.2.3 Notification with Push-Monitor

Context

A process P_1 sends a message, the notification, to be processed by a process P_2 and immediately continues execution. Although not being interested in the final result of P_2's processing, P_1 requires feedback of P_2's execution progress which is directly reported by P_2 back to P_1.

Forces

P_1 is not interested in the result of P_2's message processing. Thus, P_1 continues execution right after sending the message to P_2. In particular, both processes advance completely independent from each other. P_2 may already be busy with some execution when P_1's message arrives. This implies, that P_2 must implement at least some pseudo-parallel behaviour, enabling parallel handling of multiple notification messages.

While P_1 continues execution in parallel to P_2, P_1 wants to gain knowledge about the progress of P_2. As P_1 does not know when new status information becomes available, it is inefficient for P_1 to consecutively poll P_2. Instead, P_2 will provide status updates by itself.

Roles

- $Notifier$. Notifier sends the notification and requires status updates.
- $Notifyee$. Notifyee processes the notification message and provides status updates.

Formal Actions

- $notify()$. The action's input messages represent the notification to be communicated between notifier and notifyee. It is directed to the notifyee role.

  The formal action identifies only messages being sent from notifier to notifyee, omitting a communication channel in the opposite direction. Thus, only actions not specifying any output messages represent valid substitutes, i.e.:

$$c(notify(), a) = \begin{cases} true & \text{if } \mathcal{T}_{out}(a) = \emptyset \\ false & \text{otherwise} \end{cases}$$

- $monitor()$. The action's input messages represent execution status updates provided by notifyee. It is directed to the notifier role.

  The formal action identifies only messages being sent from notifyee to notifier, omitting a communication channel in the opposite direction. Thus, only actions not specifying any output messages represent valid substitutes, i.e.:

$$c(monitor(), a) = \begin{cases} true & \text{if } \mathcal{T}_{out}(a) = \emptyset \\ false & \text{otherwise} \end{cases}$$

Formal Specification

$$\begin{aligned} IT_{NotificationPushMonitor} = (\mathcal{R}_{it} &= \{notifier, notifyee\}, \\ \mathcal{A}_f &= \{notify(), monitor()\}, \\ D &= \{(notifyee, notify()), \\ &\quad (notifier, monitor())\}, \\ c(f, a) &= \begin{cases} true & \text{if } \mathcal{T}_{out}(a) = \emptyset \\ false & \text{otherwise} \end{cases}, \\ B_{it} &= \uparrow NotificationPushMonitor) \end{aligned}$$

Figure 7.5: Formal specification of the Notification with Push-Monitor IT.

UML Model

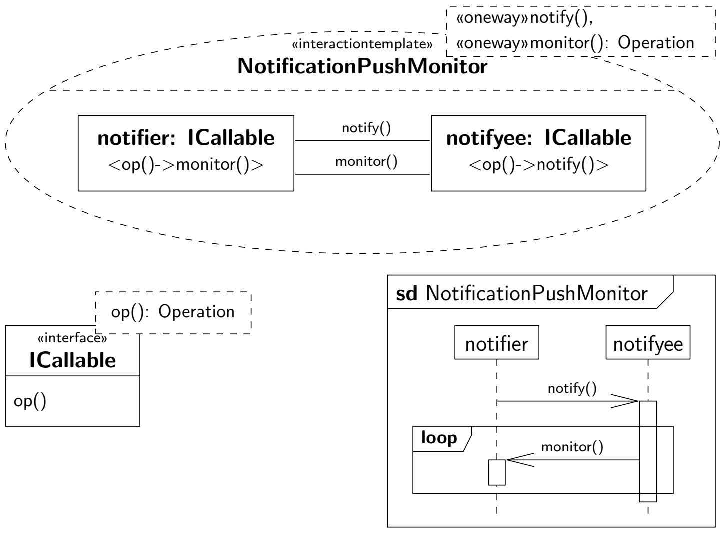


Figure 7.6: UML model of the Notification with Push-Monitor IT.

Target Mappings

Formal Action	Java/Java RMI Mapping	CORBA Mapping
$notify()$	detached Java method	IDL oneway operation
$monitor()$	detached Java method	IDL oneway operation

Table 7.4: Target mapping for the Notification with Push-Monitor IT.

7.2.4 Notification with Pull-Monitor

Context

A process P_1 sends a message, the notification, to be processed by a process P_2 and immediately continues execution. Although not being interested in the final result of P_2's processing, P_1 requires feedback of P_2's execution progress which can be pulled by P_1 from P_2. Note, that in the described flavour of this pattern, it remains unspecified for how long P_2's progress information can be pulled by P_1, especially after P_2 finished its processing.

Forces

P_1 is not interested in the result of P_2's message processing. Thus, P_1 continues execution right after sending the message to P_2. In particular, both processes advance completely independent from each other. P_2 may already be busy with some execution when P_1's message arrives. This implies, that P_2 must implement at least some pseudo-parallel behaviour, enabling parallel handling of multiple notification messages.

While P_1 continues execution in parallel to P_2, P_1 wants to gain knowledge about the progress of P_2. P_1 is not able to handle such status updates at arbitrary moments of execution time. Hence, it is more effective for P_1 to pull these updates from P_2.

Roles

- $Notifier$. Notifier sends the notification and requests status updates from notifyee.
- $Notifyee$. Notifyee processes the notification message and answers status requests.

Formal Actions

- $notify()$. The action's input messages represent the notification to be communicated between notifier and notifyee. It is directed to the notifyee role.

  The formal action identifies only messages being sent from notifier to notifyee, omitting a communication channel in the opposite direction. Thus, only actions not specifying any output messages represent valid substitutes, i.e.:

$$c(notify(), a) = \begin{cases} true & \text{if } \mathcal{T}_{out}(a) = \emptyset \\ false & \text{otherwise} \end{cases}$$

- $monitor()$. The action's input messages represent execution status request by notifier, its output message the progress information available from notifyee. It is directed to the notifyee role.

  The action allows substitution by actual actions defining at least one output message to communicate the requested status of the notifyee, i.e.:

$$c(monitor(), a) = true \begin{cases} true & \text{if } \mathcal{T}_{out}(a) \neq \emptyset \\ false & \text{otherwise} \end{cases}$$

Formal Specification

$$\begin{array}{lclcl} IT_{NotificationPullMonitor} & = & (\mathcal{R}_{it} & = & \{notifier, notifyee\}, \\ & & \mathcal{A}_f & = & \{notify(), monitor()\}, \\ & & D & = & \{(notifyee, notify()), \\ & & & & (notifyee, monitor())\}, \\ & & c(f,a) & = & \begin{cases} true & \text{if } f = notify() \wedge \mathcal{T}_{out}(a) = \emptyset \\ & \text{or } f = monitor() \wedge \mathcal{T}_{out}(a) \neq \emptyset \\ false & \text{otherwise} \end{cases}, \\ & & B_{it} & = & \upharpoonright NotificationPullMonitor) \end{array}$$

Figure 7.7: Formal specification of the Notification with Pull-Monitor IT.

UML Model

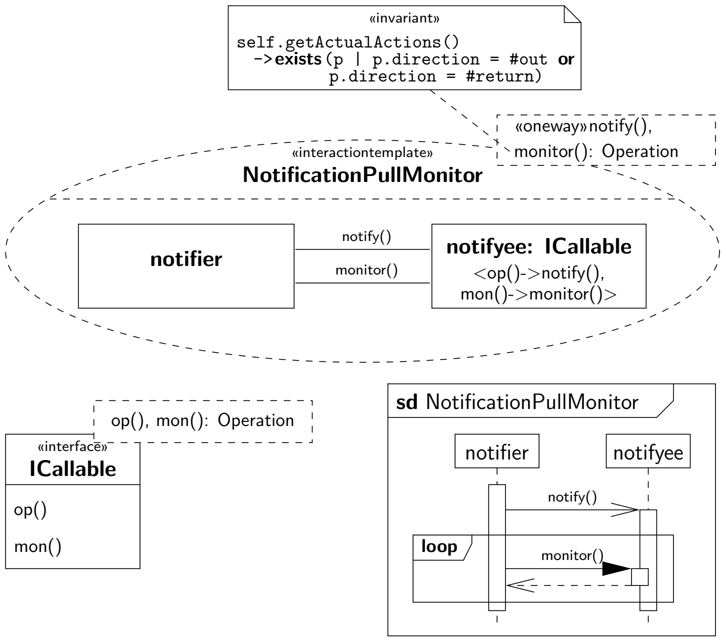


Figure 7.8: UML model of the Notification with Pull-Monitor IT.

Target Mappings

Formal Action	Java/Java RMI Mapping	CORBA Mapping
$notify()$	detached Java method	IDL oneway operation
$monitor()$	native Java method	IDL operation

Table 7.5: Target mapping for the Notification with Pull-Monitor IT.

7.2.5 Asynchronous Request/Response

Context

A process P_1 sends a message, the request, to a process P_2. P_1 may continue with other tasks while P_2 processes the message. After P_2 finished, it returns a reply message, the response, back to P_1.

Forces

If processing a message may take a significant time for P_2, it is more efficient for P_1 to continue execution rather than waiting explicitly for P_2 to finish. Hence, P_1 can pass the request asynchronously to P_2 and waits for the response while being busy with other tasks. Both processes advance completely independent from each other. Note, P_2 may already be busy with some execution when P_1's request message arrives. This implies, that P_2 must implement at least some pseudo-parallel behaviour, enabling parallel handling of multiple incoming messages. Additionally, P_1 must be able to handle the response message asynchronously to its continued execution.

Roles

- *Caller*. Caller triggers the interaction by initially sending the request message.
- *Callee*. Callee processes the request and returns the response message.

Formal Actions

- *request*(). The action's input messages represent the request. The action is directed to the callee role.

  The formal action identifies only messages being sent from caller to callee, omitting a communication channel in the opposite direction. Thus, only actions not specifying any output messages represent valid substitutes, i.e.:

$$c(request(), a) = \begin{cases} true & \text{if } \mathcal{T}_{out}(a) = \emptyset \\ false & \text{otherwise} \end{cases}$$

- *response*(). The action's input messages represent the response, i.e., the outcome of the callee's message processing. The action is directed to the caller role.

  The formal action identifies only messages being sent from callee to caller, omitting a communication channel in the opposite direction. Thus, only actions not specifying any output messages represent valid substitutes, i.e.:

$$c(response(), a) = \begin{cases} true & \text{if } \mathcal{T}_{out}(a) = \emptyset \\ false & \text{otherwise} \end{cases}$$

Formal Specification

$$\begin{array}{rcrcl} IT_{AsyncRR} & = & (\mathcal{R}_{it} & = & \{caller, callee\}, \\ & & \mathcal{A}_f & = & \{request(), response()\}, \\ & & D & = & \{(callee, request()), \\ & & & & (caller, response())\}, \\ & & c(f,a) & = & \begin{cases} true & \text{if } \mathcal{T}_{out}(a) = \emptyset \\ false & \text{otherwise} \end{cases}, \\ & & B_{it} & = & \uparrow AsyncRR) \end{array}$$

Figure 7.9: Formal specification of the Asynchronous R/R IT.

UML Model

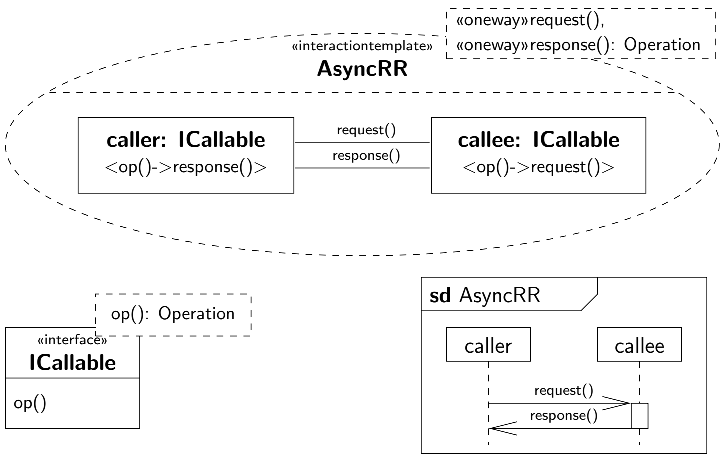


Figure 7.10: UML model of the Asynchronous R/R IT.

Target Mappings

Formal Action	Java/Java RMI Mapping	CORBA Mapping
request()	detached Java method	IDL oneway operation
response()	detached Java method	IDL oneway operation

Table 7.6: Target mapping for the Asynchronous R/R IT.

7.2.6 Asynchronous Request/Response with Push-Monitor

Context

A process P_1 sends a message, the request, to a process P_2. P_1 may continue with other tasks while P_2 processes the message. After P_2 finished, it returns a reply message, the response, back to P_1. During the time P_2 processes the request, P_1 requires feedback of P_2's execution progress which is directly reported by P_2 back to P_1.

Forces

If processing a message may take a significant time for P_2, it is more efficient for P_1 to continue execution rather than waiting explicitly for P_2 to finish. Hence, P_1 can pass the request asynchronously to P_2 and waits for the response while being busy with other tasks. Both processes advance completely independent from each other. Note, P_2 may already be busy with some execution when P_1's request message arrives. This implies, that P_2 must implement at least some pseudo-parallel behaviour, enabling parallel handling of multiple incoming messages.

While P_1 continues execution in parallel to P_2, P_1 wants to gain knowledge about the progress of P_2. As P_1 does not know when new status information becomes available, it is inefficient for P_1 to consecutively poll P_2. Instead, P_2 will provide status updates by itself. In turn, P_1 must be capable to handle these updates as well as the reply message as interruptions to its existing execution behaviour.

Roles

- *Caller*. Caller triggers the interaction by initially sending the request message and receives status updates from callee.

- *Callee*. Callee processes the request and returns the response message. Additionally, it provides asynchronous status updates to caller while processing the request.

Formal Actions

- *request*(). The action's input messages represent the request. The action is directed to the callee role.

  The formal action identifies only messages being sent from caller to callee, omitting a communication channel in the opposite direction. Thus, only actions not specifying any output messages represent valid substitutes, i.e.:

$$c(request(), a) = \begin{cases} true & \text{if } \mathcal{T}_{out}(a) = \emptyset \\ false & \text{otherwise} \end{cases}$$

- *response*(). The action's input messages represent the response, i.e., the outcome of the callee's message processing. The action is directed to the caller role.

  The formal action identifies only messages being sent from callee to caller, omitting a communication channel in the opposite direction. Thus, only actions not specifying any output messages represent valid substitutes, i.e.:

$$c(response(), a) = \begin{cases} true & \text{if } \mathcal{T}_{out}(a) = \emptyset \\ false & \text{otherwise} \end{cases}$$

- *monitor*(). The action's input messages represent execution status updates provided by callee. It is directed to the caller role.

  The formal action identifies only messages being sent from callee to caller, omitting a communication channel in the opposite direction. Thus, only actions not specifying any output messages represent valid substitutes, i.e.:

$$c(monitor(), a) = \begin{cases} true & \text{if } \mathcal{T}_{out}(a) = \emptyset \\ false & \text{otherwise} \end{cases}$$

Formal Specification

$$
\begin{aligned}
IT_{AsyncRRPushMonitor} = (\mathcal{R}_{it} &= \{caller, callee\},\\
\mathcal{A}_f &= \{request(), response(), monitor()\},\\
D &= \{(callee, request()),\\
&\quad (caller, response()),\\
&\quad (caller, monitor())\},\\
c(f,a) &= \begin{cases} true & \text{if } \mathcal{T}_{out}(a) = \emptyset \\ false & \text{otherwise} \end{cases},\\
B_{it} &= \uparrow AsyncRRPushMonitor)
\end{aligned}
$$

Figure 7.11: Formal specification of the Asynchronous R/R with Push-Monitor IT.

UML Model

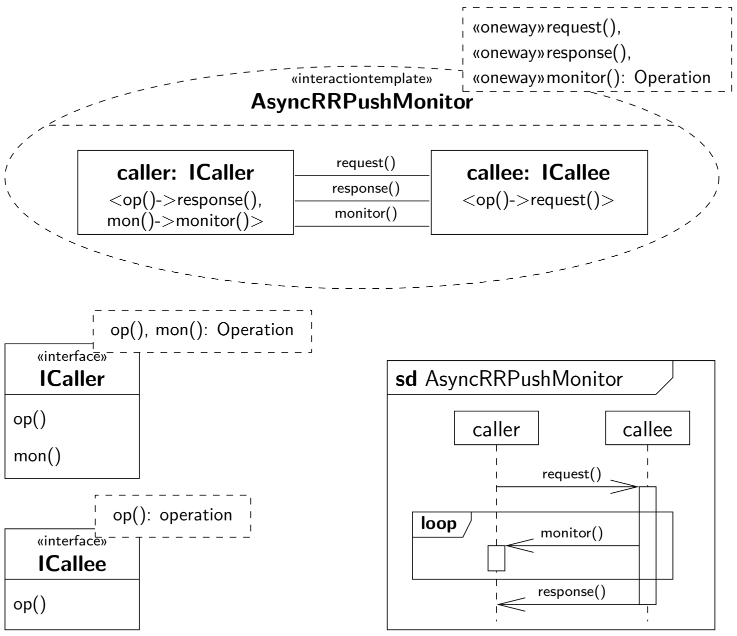


Figure 7.12: UML model of the Asynchronous R/R with Push-Monitor IT.

Target Mappings

Formal Action	Java/Java RMI Mapping	CORBA Mapping
request()	detached Java method	IDL oneway operation
response()	detached Java method	IDL oneway operation
monitor()	detached Java method	IDL oneway operation

Table 7.7: Target mapping for the Asynchronous R/R with Push-Monitor IT.

7.2.7 Asynchronous Request/Response with Pull-Monitor

Context

A process P_1 sends a message, the request, to a process P_2. P_1 may continue with other tasks while P_2 processes the message. After P_2 finished, it returns a reply message, the response, back to P_1. During the time P_2 processes the request, P_1 requires feedback of P_2's execution progress which can be pulled by P_1 from P_2. Note, that P_1 is only allowed to pull progress information from P_2 as long as P_2 has not sent the response message.

Forces

If processing a message may take a significant time for P_2, it is more efficient for P_1 to continue execution rather than waiting explicitly for P_2 to finish. Hence, P_1 can pass the request asynchronously to P_2 and waits for the response while being busy with other tasks. Both processes advance completely independent from each other. Note, P_2 may already be busy with some execution when P_1's request message arrives. This implies, that P_2 must implement at least some pseudo-parallel behaviour, enabling parallel handling of multiple incoming messages.

While P_1 continues execution in parallel to P_2, P_1 wants to gain knowledge about the progress of P_2. P_1 is not able to handle such status updates at arbitrary moments of execution time. Hence, it is more effective for P_1 to pull these updates from P_2. In consequence, P_2 must be able to handle status request asynchronously to its own processing. Note, that P_1 can only validly pull P_2's progress as long as P_2 effectively processes the request. Once finished processing, P_2 may not provide meaningful status information.

Roles

- *Caller*. Caller triggers the interaction by initially sending the request message and requests status updates from callee.
- *Callee*. Callee processes the request and returns the response message. At the same time of execution it answers status update request from caller.

Formal Actions

- *request*(). The action's input messages represent the request. The action is directed to the callee role.

  The formal action identifies only messages being sent from caller to callee, omitting a communication channel in the opposite direction. Thus, only actions not specifying any output messages represent valid substitutes, i.e.:

$$c(request(), a) = \begin{cases} true & \text{if } \mathcal{T}_{out}(a) = \emptyset \\ false & \text{otherwise} \end{cases}$$

- *response*(). The action's input messages represent the response, i.e., the outcome of the callee's message processing. The action is directed to the caller role.

The formal action identifies only messages being sent from callee to caller, omitting a communication channel in the opposite direction. Thus, only actions not specifying any output messages represent valid substitutes, i.e.:

$$c(response(), a) = \begin{cases} true & \text{if } \mathcal{T}_{out}(a) = \emptyset \\ false & \text{otherwise} \end{cases}$$

- *monitor*(). The action's input messages represent execution status request by caller, its output message the progress information available from callee. It is directed to the callee role.

  The action allows substitution by actual actions defining at least one output message to communicate the requested status of the callee, i.e.:

$$c(monitor(), a) = true \begin{cases} true & \text{if } \mathcal{T}_{out}(a) \neq \emptyset \\ false & \text{otherwise} \end{cases}$$

Formal Specification

$$\begin{aligned} IT_{AsyncRRPullMonitor} = (\mathcal{R}_{it} &= \{caller, callee\}, \\ \mathcal{A}_f &= \{request(), response(), monitor()\}, \\ D &= \{(callee, request()), \\ &\quad (callee, monitor()), \\ &\quad (caller, response())\}, \\ c(f, a) &= \begin{cases} true & \text{if } f = request() \wedge \mathcal{T}_{out}(a) = \emptyset \\ & \text{or } f = response() \wedge \mathcal{T}_{out}(a) = \emptyset \\ & \text{or } f = monitor() \wedge \mathcal{T}_{out}(a) \neq \emptyset \\ false & \text{otherwise} \end{cases}, \\ B_{it} &= \uparrow AsyncRRPullMonitor) \end{aligned}$$

Figure 7.13: Formal specification of the Asynchronous R/R with Pull-Monitor IT.

UML Model

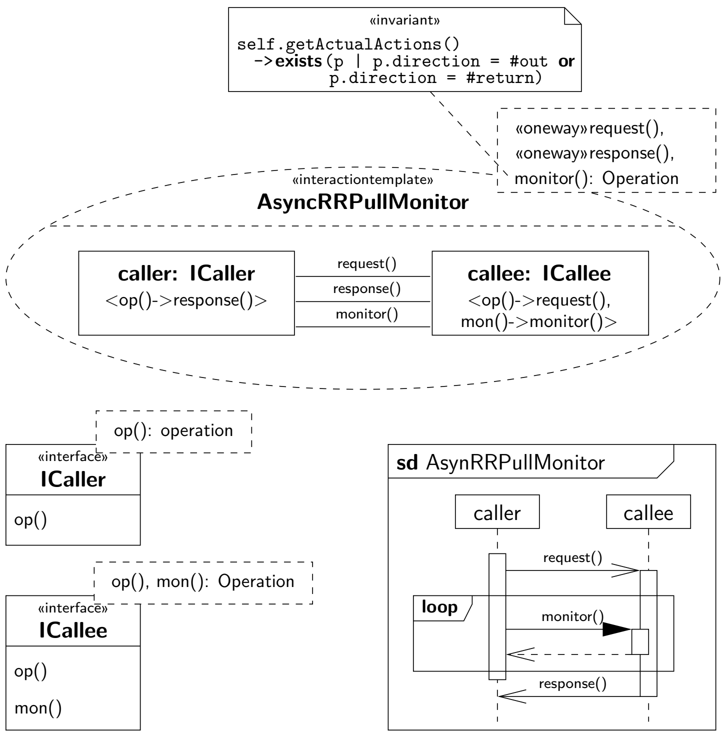


Figure 7.14: UML model of the Asynchronous R/R with Pull-Monitor IT.

Target Mappings

Formal Action	Java/Java RMI Mapping	CORBA Mapping
request()	detached Java method	IDL oneway operation
response()	detached Java method	IDL oneway operation
monitor()	native Java method	IDL operation

Table 7.8: Target mapping for the Asynchronous R/R with Push-Monitor IT.

7.2.8 Abortable Asynchronous Request/Response

Context

A process P_1 sends a message, the request, to a process P_2. P_1 may continue with other tasks while P_2 processes the message. After P_2 finished, it returns a reply message, the response, back to P_1. While P_2 processes the message, P_1 may choose to abort P_2's task instead of waiting for a response message. In the case of cancellation, P_2 provides feedback about the effects of aborting its processing.

Forces

If processing a message may take a significant time for P_2, it is more efficient for P_1 to continue execution rather than waiting explicitly for P_2 to finish. Hence, P_1 can pass the request asynchronously to P_2 and waits for the response while being busy with other tasks. Both processes advance completely independent from each other. Note, P_2 may already be busy with some execution when P_1's request message arrives. This implies, that P_2 must implement at least some pseudo-parallel behaviour, enabling parallel handling of multiple incoming messages.

While P_1 continues execution in parallel to P_2, it may decide to cancel the request for P_2. Aborting a command may involve complex intermediary tasks for P_2, e.g., cleaning up partial results. Although, these steps are not considered as part of the IT, P_1 blocks until P_2 provides feedback about the handling of the abort request. Note, P_1 can only invoke the abort command prior to receive the response message. Furthermore, once aborted, P_2 must not provide a delayed response for the cancelled request.

Receiving a response message or aborting a command are mutually exclusive interactions. A target adaptor for this IT may include wrapper code to opaquely handle race conditions between both interactions for the involved processes.

Roles

- *Caller*. Caller triggers the interaction by initially sending the request message and may abort message processing of callee.
- *Callee*. Callee processes the request and returns the response message. While processing the request message, callee may receive an abort command, cancelling execution.

Formal Actions

- *request*(). The action's input messages represent the request. The action is directed to the callee role.

  The formal action identifies only messages being sent from caller to callee, omitting a communication channel in the opposite direction. Thus, only actions not specifying any output messages represent valid substitutes, i.e.:

$$c(request(), a) = \begin{cases} true & \text{if } \mathcal{T}_{out}(a) = \emptyset \\ false & \text{otherwise} \end{cases}$$

- *response*(). The action's input messages represent the response, i.e., the outcome of the callee's message processing. The action is directed to the caller role.

  The formal action identifies only messages being sent from callee to caller, omitting a communication channel in the opposite direction. Thus, only actions not specifying any output messages represent valid substitutes, i.e.:

$$c(response(), a) = \begin{cases} true & \text{if } \mathcal{T}_{out}(a) = \emptyset \\ false & \text{otherwise} \end{cases}$$

- *abort*(). The action's input messages represent the abort request of caller, its output messages the cancellation feedback from callee, respectively. The action is directed to the callee role.

  The action allows substitution by arbitrary actual actions, i.e.:

$$c(abort(), a) = true$$

Formal Specification

$$\begin{array}{llll} IT_{AsyncRRPullMonitor} & = & (\mathcal{R}_{it} & = \{caller, callee\}, \\ & & \mathcal{A}_f & = \{request(), response(), abort()\}, \\ & & D & = \{(callee, request()), \\ & & & \quad (callee, abort()), \\ & & & \quad (caller, response())\}, \\ & & c(f,a) & = \begin{cases} true & \text{if } f = abort() \vee \mathcal{T}_{out}(a) = \emptyset \\ false & \text{otherwise} \end{cases}, \\ & & B_{it} & = \uparrow AsyncRRPullMonitor) \end{array}$$

Figure 7.15: Formal specification of the abortable Asynchronous R/R IT.

UML Model

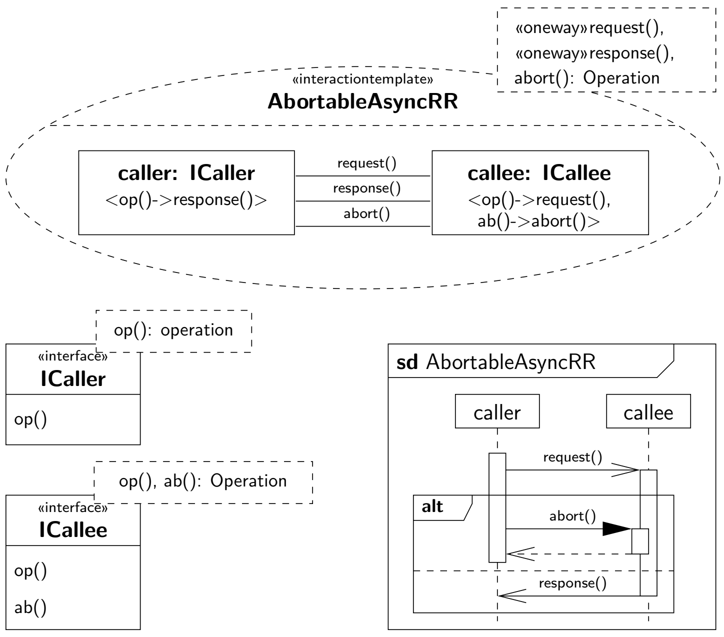


Figure 7.16: UML model of the abortable Asynchronous R/R IT.

Target Mappings

Formal Action	Java/Java RMI Mapping	CORBA Mapping
request()	detached Java method	IDL oneway operation
response()	detached Java method	IDL oneway operation
abort()	native Java method	IDL operation

Table 7.9: Target mapping for the Asynchronous R/R with Push-Monitor IT.

7.3 Summary

This chapter showed the application of the service development process to create an IT library to serve as basis for service and system specification. The library demonstrated the modelling activities necessary to describe individual ITs, exemplified by capturing primitive as well as complex interaction patterns.

CHAPTER 8

Defining Services and Systems

In this chapter, we present two examples of service definitions based on the IT library from Section 7.2. The system design goal is to develop a simple video processing system which captures video data from a camera and compresses selected images for storage. On the one hand, the *video capturing service* models the necessary interactions to acquire images from a camera. On the other hand, the *image compression service* models the necessary steps to compress these images. The following sections describe both services and the resulting system in detail, illustrating the tasks of the service and system designer. Note, that a service identifies the number and kinds of interactions which occur between its roles. It neither defines internal behaviour of the roles nor does it specify how the interactions are inter-weaved. The same applies for the service components as elements of the composed system. Therefore, a system specification identifies services as "glue" in-between components, considering the concrete functionality of a component as opaque. In this sense, a component definition is given by the aggregation of the service roles it is connected to.

8.1 Example Services

8.1.1 The Video Capturing Service

The video capturing service provides means to retrieve video data from a camera. Therefore, the service identifies two roles:

Camera. The camera role represents the source for video data. The reader may imagine a real, physical camera or a playback system implementing this role.

Client. The client role represents the sink video data is delivered to. For instance, this role is fulfilled by a video recording system or a monitor.

Both roles, camera and client, communicate by three interactions. Two interactions are used as control commands to the camera's state and one is exploited to transfer video data:

Activate. The activate interaction is used by the client to turn on the camera system. As the client expects immediate feedback if the camera was activated correctly, this interaction is based on the Synchronous R/R IT (Section 7.2.1). The IT's formal $request()$ action is substituted by the actual action $start() : Boolean$. The action's Boolean output message is used to signal either success or failure of the activation command. Formally, this interaction is given by:

$$\begin{aligned} activate = (IT &= IT_{SyncRR}, \\ \mathcal{A} &= \{(request(), start() : Boolean)\} \quad) \end{aligned}$$

Deactivate. The deactivate interaction is the counter part to the previous interaction. It is initiated by the client to gracefully shut down the camera. Similar to activate, this interaction is also based on the Synchronous R/R IT (Section 7.2.1) with an analog action replacement:

$$\begin{aligned} deactivate = (IT &= IT_{SyncRR}, \\ \mathcal{A} &= \{(request(), stop() : Boolean)\} \quad) \end{aligned}$$

Stream. The last interaction of the video capturing service is the stream interaction. It is used by the camera to transmit video images to the client. As the camera does not need any feedback from the client upon the reception of an image, this interaction is based on the Notification IT (Section 7.2.2). Hereby, each image is sent with an integer time-stamp, enabling the client to correlate the individual images after reception. The stream interaction is formally defined as:

$$\begin{aligned} stream = (IT &= IT_{Notification}, \\ \mathcal{A} &= \{(notify(), nextImage(Integer, Image))\} \quad) \end{aligned}$$

Figure 8.1 depicts both, the formal specification (a) and the UML model (b) of the video capturing service. As defined by the role mapping relation tuple $\mathcal{M}$, the two control interactions the ITs' caller roles are bound to the service's client role, as the client triggers these interactions. In turn, the ITs's callee roles are linked to the camera role. For the third interaction, having contrary responsibilities for the chain of control, the camera plays the role of the IT's notifier role and the client becomes the notifyee.

8.1.2 The Image Compression Service

The image compression service provides functionality to compress images. The service identifies two roles:

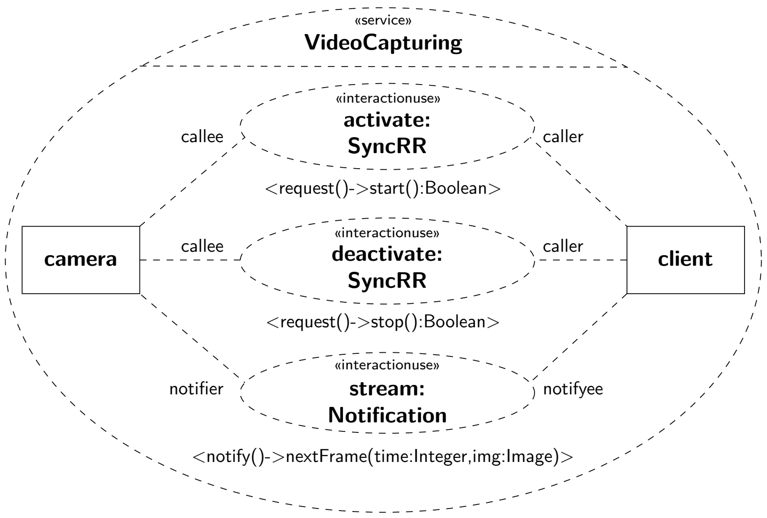
Codec. The codec role will implement the compression algorithm for video images.

Client. The client role provides the raw image data to be compressed by the codec and receives back a condensed version.

Both roles, codec and client, are connected by three interactions. Two interactions support querying or manipulating the codec's configuration and one interaction is used to exchange raw and compressed image data. As image compression may be a "long" running task executed in separate phases, e.g., analysis, quantisation, or vectorisation, the latter interaction provides means for monitoring the progress of the compression. In detail, the service's interactions are:

$$
\begin{aligned}
Svc_{VideoCapturing} \;=\; (\mathcal{R}_{svc} &= \{camera, client\} \\
\mathcal{I}_{inst} &= ((IT = IT_{SyncRR}, \\
&\quad \mathcal{A} = \{(request(), start() : Boolean))\}, \\
&\quad (IT = IT_{SyncRR}, \\
&\quad \mathcal{A} = \{(request(), stop() : Boolean))\}, \\
&\quad (IT = IT_{Notification}, \\
&\quad \mathcal{A} = \{(notify(), nextImage(Integer, Image))\})), \\
\mathcal{M} &= (\{(caller, client), (callee, camera)\}, \\
&\quad \{(caller, client), (callee, camera)\}, \\
&\quad \{(notifier, camera), (notifyee, client)\}) \quad)
\end{aligned}
$$

(a) Formal specification.



(b) UML model.

Figure 8.1: The video capturing service.

Configure. This interaction is used by the client to set up the codec with the right properties. In turn, the codec provides feedback about the validity of the requested compression mode. Hence, the configure interaction is based on the Synchronous R/R IT (Section 7.2.1). The formal $request()$ action is substituted by $setMode(CMode) : Boolean$ thereby $CMode$ represents the message type to describe the requested compression mode. The boolean return message indicates successful configuration. Formally, this interaction is given by:

$$\begin{aligned} configure \; = \; (IT \; &= \; IT_{SyncRR}, \\ \mathcal{A} \; &= \; \{(request(), setMode(CMode) : Boolean)\} \quad) \end{aligned}$$

Retrieve. The retrieve interaction is the complement to configure. It is invoked by the client to fetch the current compression configuration from the codec. Thus, as the client pulls this information from the codec, the interaction is based on the Synchronous R/R IT (Section 7.2.1) which natively supports a backward channel. There is no input message defined for this action, leading to the following formal definition:

$$\begin{aligned} retrieve \; = \; (IT \; &= \; IT_{SyncRR}, \\ \mathcal{A} \; &= \; \{(request(), getMode() : CMode)\} \quad) \end{aligned}$$

Compress. Finally, the compress interaction is used by the client to actually request the compression of an image. As already mentioned above, this process may be "long" running. Because of this, the interaction is asynchronous, enabling the client to continue its own execution without having to wait for the codec to finish. However, the client will still automatically receive progress updates (in percent) about the codec internal progress via an interaction monitor. Consequently, this IT is based on the Asynchronous R/R IT with Push Monitor (Section 7.2.6) and is formally specified as:

$$\begin{aligned} compress \; = \; (IT \; &= \; IT_{AyncRRPushMonitor}, \\ \mathcal{A} \; &= \; \{(request(), compress(Image)), \\ & \quad\; (response(), compressed(CImage)), \\ & \quad\; (monitor(), progress(Integer))\} \quad) \end{aligned}$$

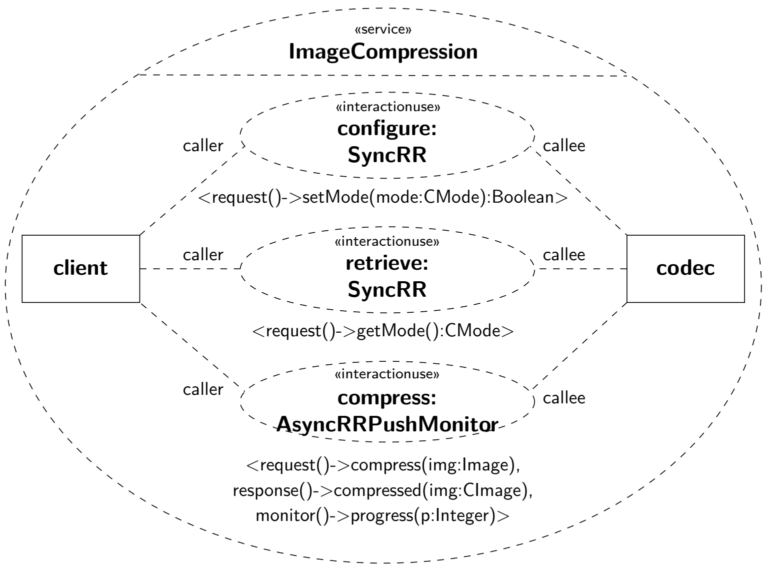
The service is presented in Figure 8.2 including its formal specification (a) and its illustration as UML model (b). As the configure and retrieve interactions are used by the client to control the codec and as such are initiated by the client, the underlying Synchronous R/R ITs' caller role is bound to the client, the callee role to the codec, as stated by the role binding relation tuple $\mathcal{M}$. The same role binding is also established for the compress interaction's IT.

8.2 The Video Recording System

We use the video capturing and the image compression services to construct a simple video recording system. The system does not describe a real world video recorder. Its purpose is to demonstrate how services are used to specify systems, i.e., to connect service components and how such specifications drive our model transformation processes. Thus, we omit detailed information about the concrete functionalities of each component. The video recording system consists of three components:

$$
\begin{aligned}
Svc_{ImageCompression} \;=\; (\mathcal{R}_{svc} &= \{client, codec\} \\
\mathcal{I}_{inst} &= ((IT = IT_{SyncRR}, \\
&\quad\;\; \mathcal{A} = \{(request(), setMode(CMode) : Boolean))\}, \\
&\quad (IT = IT_{SyncRR}, \\
&\quad\;\; \mathcal{A} = \{(request(), getMode() : CMode))\}, \\
&\quad (IT = IT_{AsyncRRPushMonitor}, \\
&\quad\;\; \mathcal{A} = \{(request(), compress(Image)), \\
&\qquad\qquad (response(), compressed(CImage)), \\
&\qquad\qquad (monitor(), progress(Integer))\})), \\
\mathcal{M} &= (\{(caller, client), (callee, codec)\}, \\
&\quad\; \{(caller, client), (callee, codec)\}, \\
&\quad\; \{(caller, client), (callee, codec)\}) \quad)
\end{aligned}
$$

(a) Formal specification.



(b) UML model.

Figure 8.2: The image compression service.

$$
\begin{aligned}
Comp_{VideoCamera} = (id &= VideoCamera,\\
Ports &= \{(\mathcal{S} = \uparrow Svc_{VideoCapturing},\\
&\qquad \mathcal{R} = camera)\} \quad)
\end{aligned}
$$

(a) The video camera component.

$$
\begin{aligned}
Comp_{Recorder} = (id &= Recorder,\\
Ports &= \{(\mathcal{S} = \uparrow Svc_{VideoCapturing},\\
&\qquad \mathcal{R} = client),\\
&\quad (\mathcal{S} = \uparrow Svc_{ImageCompression}),\\
&\qquad \mathcal{R} = client)\} \quad)
\end{aligned}
$$

(b) The recorder component.

$$
\begin{aligned}
Comp_{JP2KCompressor} = (id &= JP2KCompressor,\\
Ports &= \{(\mathcal{S} = \uparrow Svc_{ImageCompression},\\
&\qquad \mathcal{R} = codec)\} \quad)
\end{aligned}
$$

(c) The JP2K compressor component.

Figure 8.3: Formal specification of the video recording system components.

Video Camera. The video camera component functions as sensor and thus provides input to the system.

Recorder. The recorder component receives video data from the camera and stores compressed images of this video.

JP2K Compressor. The compressor component implements an image compression algorithm. For illustration, the component will provide an *Joint Photographic Experts Group* (*JPEG*) encoder for *JPEG 2000* (*JP2K*) [ISO04].

The video camera and the recorder component establish the video capturing service. Thereby, the recorder plays the client role, the video camera the camera service role respectively. Next, the image compression service is used to connect the recorder to the JP2K compressor where again the recorder is linked to the client role and the compressor fulfils the codec role. This configuration leads to the formal definitions of the components as provided in Figure 8.3. For simplification within the figure, the notation "$\uparrow$*Service*" is used to refer to the definition of the named service as introduced in the previous sections. Note, that the recorder component defines two ports as it participates in two services. The resulting system model in UML is shown by Figure 8.4.

8.3 Summary

Chapter 8 demonstrated the application of the service development process for service and system specification to design a simple video recording system. The system consists of three individual components, a video camera, an image compressor, and a recorder. The components are interconnected by two separate services, the video capturing and the image compression service. The services were composed out of interactions derived from the IT

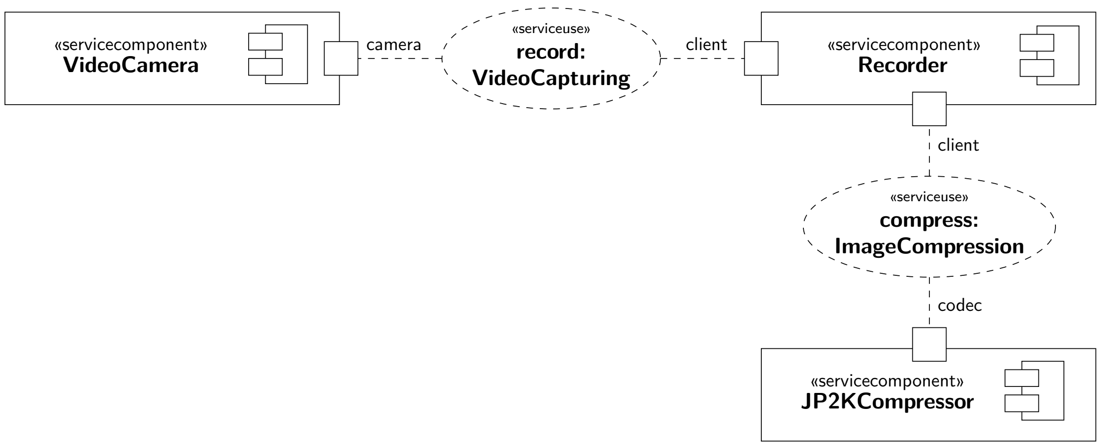


Figure 8.4: UML model of the video recording system.

library. Thereby, the service models and thus, the system specification, stayed independent from possible target platforms.

CHAPTER 9

Applying Model Transformations

This chapter describes the necessary model transformations applied to service PIMs leading to service component implementations and to target adaptors for the services' interactions. The individual transformations are exemplified for the video recording system and its both services, the video capturing service as well as the image compression service, introduced in Chapter 8.

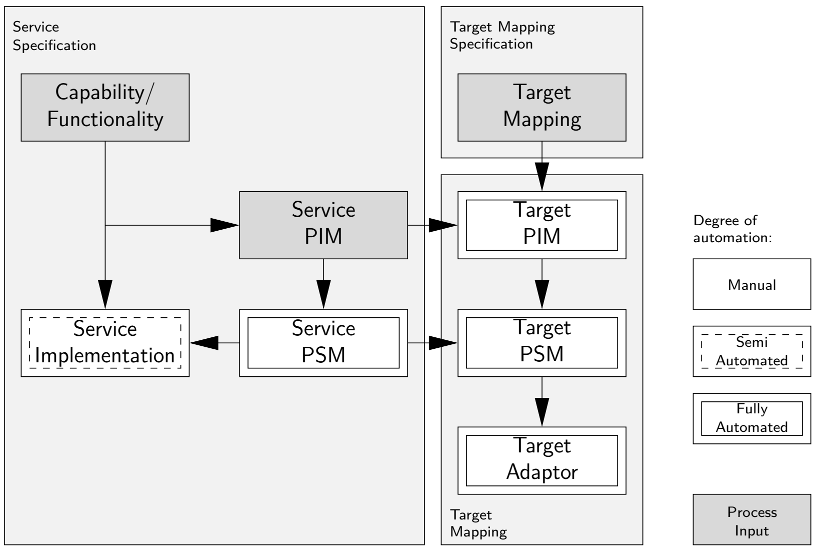


Figure 9.1: Transformational steps in the service development process (cf. Figure 5.2, p. 50).

Figure 9.1 depicts the relevant parts for model model transformations within the proposed service development process as introduced in Section 5.1. The following sections describe the individual steps in more detail. The transformation in Section 9.1 forms a common grounding for all further model transformation activities by making the components' interfaces explicit, deriving them from the service PIM. Next, in Section 9.2, we show how the service PSM results from the service PIM and how this model supports service implementation by code generation. Orthogonal to these activities, the target PIMs, PSMs and eventually the target adaptors are derived from the service PIMs/PSMs, described in Section 9.3.

9.1 Component Interfaces

A component is connected to services via ports. For each such port, we can derive provided and required interfaces. These interfaces represent the contributions of the component to the interactions of the service role its port is bound to (cf. Section 5.2.4). The interfaces are expressed by sets of actual actions associated to the corresponding interaction role by either the role is a destination of (provided) or a source of (required) this action. These interfaces form the functional closure of a component and its interactions. Thus, they are *the* essential elements to be taken into account when implementing components and realising interactions in target adaptors. Therefore, we show how a component's interfaces are derived from the component model as a common step to all following model transformation activities. The outcome of these activities is the *interface model*. The interface model is still considered as part of the service PIM although it is not modelled by the service designer himself. However, it represents no additional knowledge with respect to the service PIM but makes inherent information about components and port types more explicit.

As stated in Definition 8 and 9 (p. 64), each port of a component has a number of provided and required interfaces according to the number of interactions the port's service role is connected to. These interfaces are derived from the port's formal specification. Continuing the example of the video recording system introduced in the previous chapter, Table 9.1 shows the interfaces for the video camera, the recorder, and the JP2K compressor component, respectively. The table also illustrates the underlying "backtracking" processes: The sole service and service role binding of the port (columns 1 and 2) is split for each interaction and thus interaction role the service role is connected to (columns 3 and 4). According to the afore mentioned definitions, for each interaction role we can determine the set of actual actions the role is a destination of (column 5) or a source of (column 6). Hereby, the former case represents the provided interface and the latter the required one for the interaction role. Note, that as we use binary services, i.e., services with only two participants, the recorder component has exactly the complementary interface specification than the other two components, the video camera and the JP2K compressor component respectively, as it combines both services through its ports.

Similar to their formal pendants, we can formulate Definition 8 and 9 as model transformation rules operating on the UML representations of components, services, interactions, and ITs to reflect the interfaces as UML models. In UML, we use an analog backtracking mechanism leading from a service role back to the interface templates used as type definitions for interaction roles within ITs. In particular, the following steps are necessary:

1. Determine the service role of a port. The service is given by the «serviceuse» collaboration use's role binding connected to the port. This step corresponds to columns 1 and 2 in Table 9.1.

2. Determine the interaction roles connected to the service role. This information is provided via the owning «service» collaboration of the service role. Within this collaboration, we can query for the «interactionuse» collaboration uses binding the service

Port (p)		Interaction		Interface	
Service $\mathcal{S}(p)$	Role $\mathcal{R}(p)$	Interaction (i)	Role $role_{it}(s, p, i)$	Provided $provided(p)$	Required $required(p)$
Video Camera Component					
Video Capturing	*camera*	activate	*callee*	$\{start() : Boolean\}$	
		deactivate	*callee*	$\{stop() : Boolean\}$	
		stream	*caller*		$\{nextFrame(Integer, Image)\}$
JP2K Compressor Component					
Image Compression	*codec*	configure	*callee*	$\{setMode(CMode) : Boolean\}$	
		retrieve	*callee*	$\{getMode() : CMode\}$	
		compress	*callee*	$\{compress(Image)\}$	$\{compressed(CImage), progress(Integer)\}$
Recorder Component					
Video Capturing	*client*	activate	*caller*		$\{start()\}$
		deactivate	*caller*		$\{stop()\}$
		stream	*callee*	$\{nextFrame(Integer, Image)\}$	
Image Compression	*client*	configure	*caller*		$\{setMode(CMode) : Boolean\}$
		retrieve	*caller*		$\{getMode() : CMode\}$
		compress	*caller*	$\{compressed(CImage), progress(Integer)\}$	$\{compress(Image)\}$

Table 9.1: The provided and required interfaces for the sample components per service and interaction role, respectively (cf. Definition 8 and 9, p. 64).

role. This query returns the interaction roles themselves as well as the containing «interaction» collaborations. This step corresponds to columns 3 and 4 in Table 9.1.

3. For each interaction role, determine the interface used to type this role. Note, that the «interaction» collaboration represents a template instantiation of an «interaction-template» collaboration and that the roles of this collaboration are explicitly typed by interfaces which group a roles *provided* actions/operations and thus expressing the "is-destination-of" relation D of an IT (cf. Section 5.2.1, p. 57). A collaboration role of an interaction model inherits this typing information upon template instantiation. Hence, we can directly use this type definition as provided interface for this role. This step corresponds to column 5 in Table 9.1.

4. For each interaction role, determine the interface used to type the complementary interaction role. Basically, this is the same query as in Step 3 but using the type definition of the "other" collaboration role of the interaction model as this models the "is-source-of" relation $\overline{D}$ of an IT. This "other" role is well defined because an interaction specifies exactly two roles. This step corresponds to column 6 in Table 9.1.

Figure 9.2 illustrates these steps for the port of the video camera component from Section 8.2. The port is connected to the camera role of the video capturing service. For clarity, only the stream interaction of the service is depicted. Within this interaction, the camera service role plays the notifier role. The notifier interaction role is not typed, i.e., is has no action of which it is a destination of. This results in an empty *provided* interface (INotifier) for the camera service role within the stream interaction. The complementary interaction role to notifier is notifyee which is typed through an instantiation of the ICallable interface template, offering the nextFrame(. . .) action/operation. In consequence this derived interface, denoted as INotifyee, becomes a *required* interface for the camera service role.

Next, the described idea for interface determination is embedded in a complete model transformation which extracts all provided and required interfaces for a system/component specification, creating the interface model. The result of this transformation is a model consisting of a dedicated UML package for each referenced service. In turn, these service packages contain a number of sub-packages, one for each interaction of a service. The interaction packages group the interfaces for both interaction roles, defining the provided actions/operations for each interaction role. Furthermore, the surrounding service package includes an interface for each service role which is derived from the associated interaction role interfaces by a UML generalisation association. Finally, these service role interfaces are linked to the respective component ports as either provided or required interface definition. Figure 9.3 depicts the resulting interface model of the video recording system, reflecting the information formally given in Table 9.1 in UML notation.

9.2 Supporting Service Implementation

The interface model is furthermore refined to derive the service PSM. The service PSM reflects the interface model, and thus the service PIM, in the context of a programming language. This PSM is basically the same as the interface model but adds necessary information for the implementation of service components or to execute such components in specific run-time environments. First of all, the service PSM will apply stronger constraints

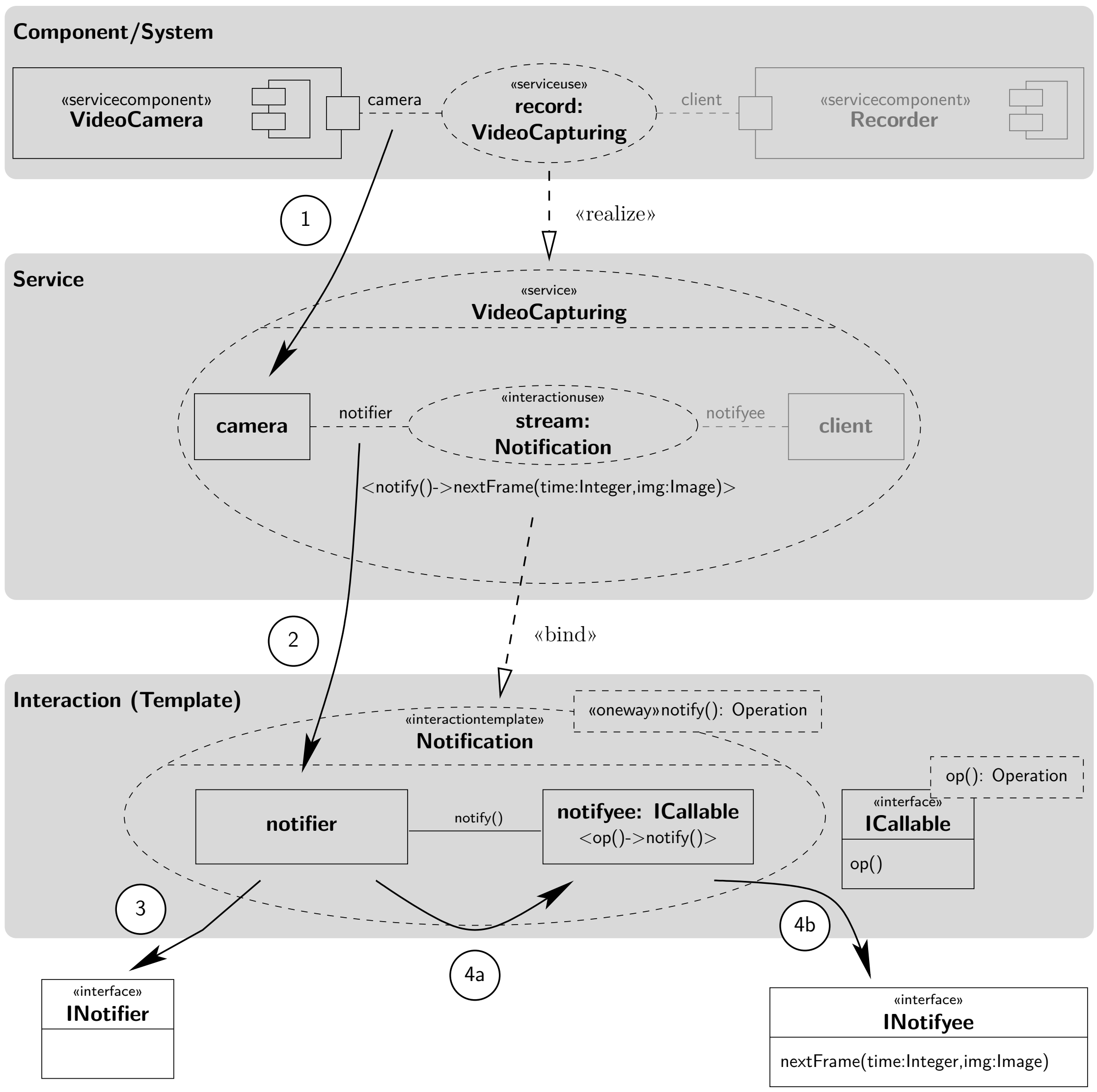


Figure 9.2: Determining a port's interfaces from UML models for the video camera component: (1) determine a port's service role, (2) determine connected interaction roles[a], (3) extract provided interfaces, and (4) extract required interfaces ((4a): determine the complementary interaction role; (4b): determine the concrete interface used to type this role).

[a]Note, the video capturing service is simplified for illustration purposes.

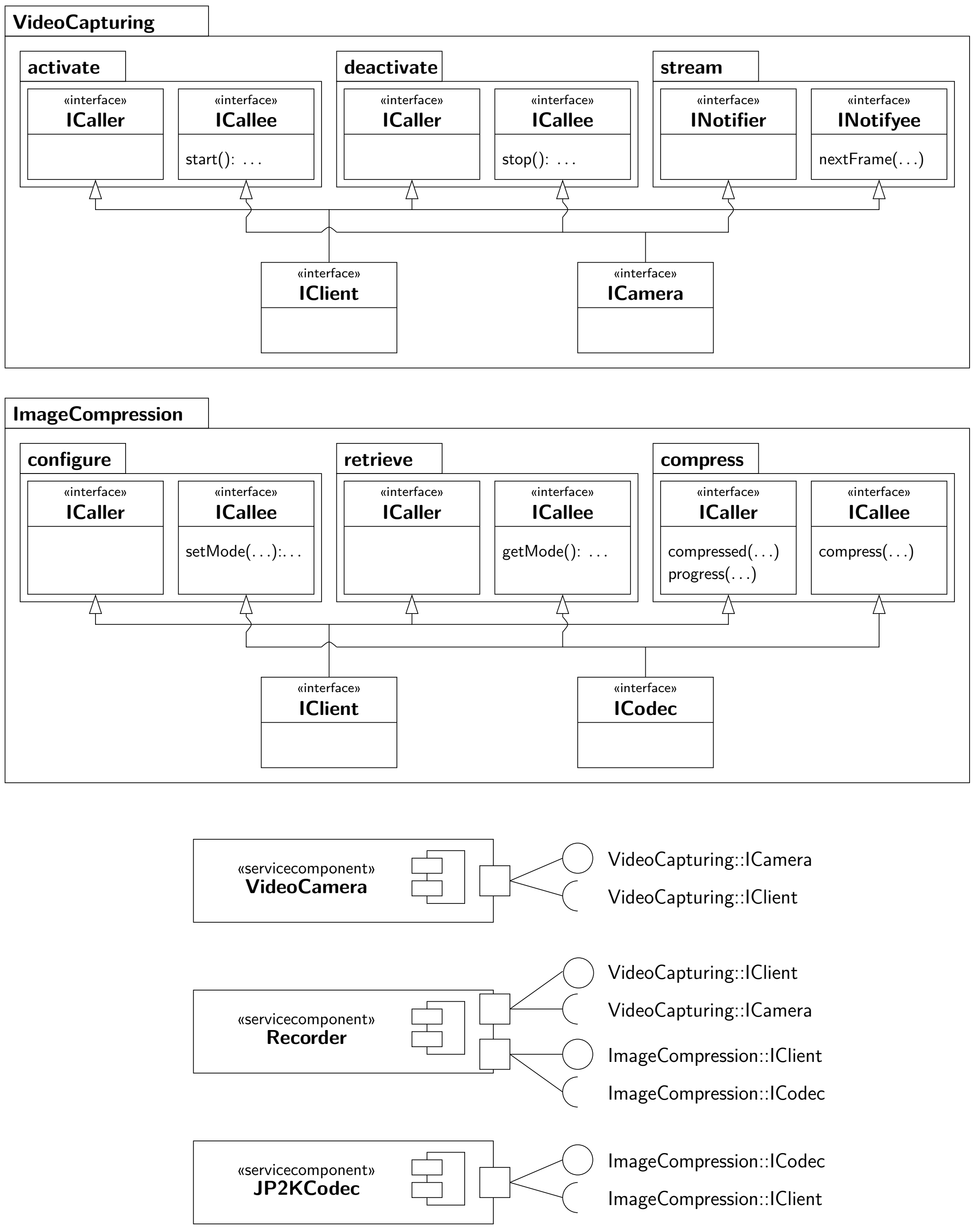


Figure 9.3: Provided and required interfaces for the video recording system's components as UML interface model (operation parameters omitted).

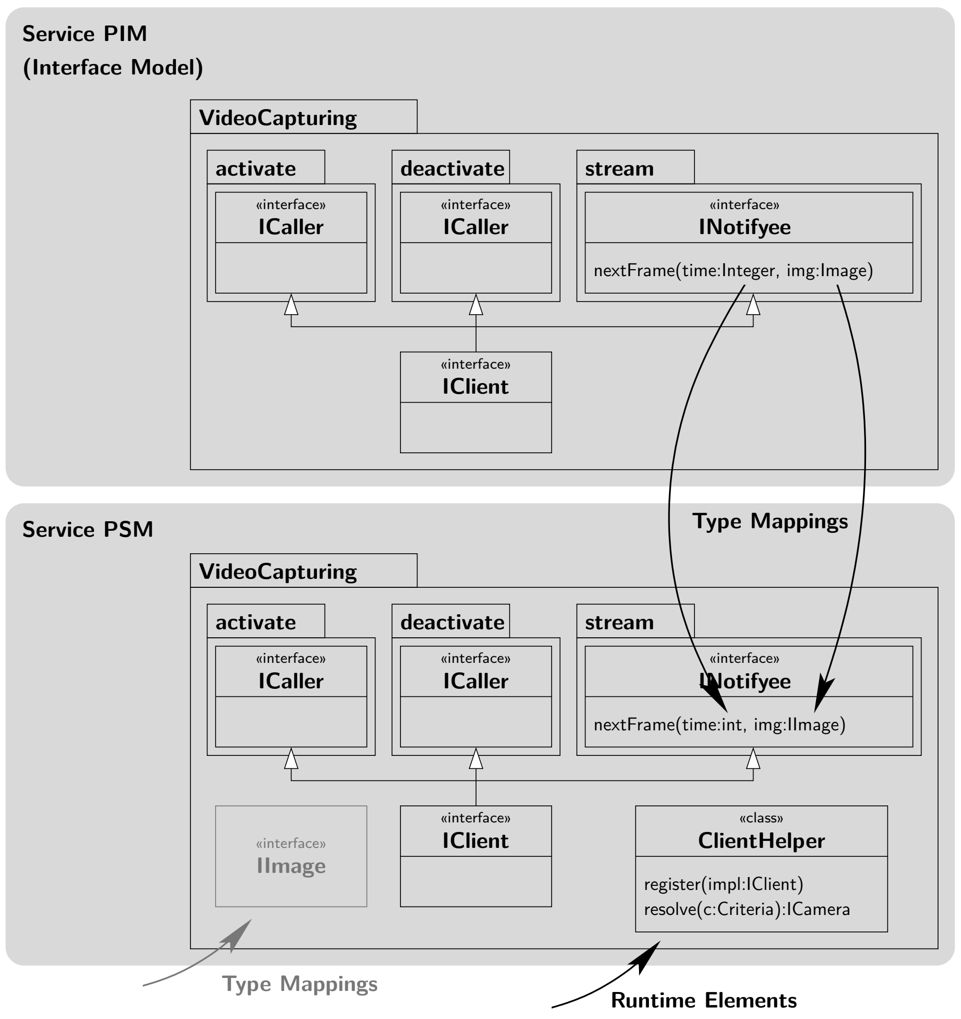


Figure 9.4: Partial service PSM of the video capturing service (reduced to client service role).

to the model elements originating from the programming languages, e.g., forbid multiple inheritance. Second, elements from the run-time environment may be added to the model, e.g., the use/generation of helper classes for service role registration and discovery. And third, the message types have to be mapped to primitives or complex constructs of the programming language. Note, that the concrete handling of message types remains out of scope of this thesis as it does not affect our proposed design process. Therefore, a detailed discussion accompanying the sample use case is omitted for this section. However, the interested reader is referred to Section D.2 in Appendix D on page 163 for a short outline of how message types are handled within the case study.

The resulting service PSM is then used as input for an M2T transformation. This transformation emits stub/skeleton code based on the UML model which enables the implementation of service components (cf. Appendix F on page 171 for the Java interfaces resulting from the presented example).

Figure 9.4 depicts an excerpt of the service PSM for our video recording system and how it is related to the interface model. The model shows the client role of the video capturing service and its associated interaction roles. The PSM is for the Java programming language. As the mapping of UML class diagrams to Java is intuitively, no additional constraints are applied to the model. The parameter types of the nextFrame(...) operation are mapped to Java types[1] and the ClientHelper class is added to the model. This helper class represents a run-time specific element which provides means to register a client service role implementation with the run-time environment[2]. Additionally, the helper is used by the service developer to query for camera implementations as interaction partner for the client, based on some (hereby opaque) selection criteria. Note, the depicted service PSM just exemplifies the necessary model refinement compared to the service PIM. It does not represent a normative reference for such models or run-time systems as the service PSM heavily depends on the concrete programming language as well as the run-time environment.

The generated source code will directly reflect the package structure, interfaces, and classes of the service PSM. The developer implements the appropriate service role interfaces and uses the mentioned helper classes to interact with the service run-time. As he only uses the service roles' interfaces for component implementation, he will not deal with the interaction roles' interfaces themselves. Thus, he is not concerned with any details about the realisation of the underlying interactions.

9.3 Generating Target Adaptors

Orthogonal to implementing the service roles in components, the service's interactions have to be realised on concrete target platforms. For illustration, we describe such a target mapping based for CORBA using the stream interaction of the video capturing service.

First, a target PIM is created. This model describes, how the interactions of a service are represented on a communication target platform, e.g., CORBA. The target PIM is derived from the service PIM as depicted in Figure 9.5 based on the rules defined in the target mapping specification (illustrated by the arrows). The interaction role interfaces are converted to technology specific representations. For instance, the INotifyee interface in the stream package becomes a CORBA interface within a corresponding CORBA module. For every action bound to an interaction role, we query the underlying communication semantics from the IT the interaction is derived from. Then, these semantics are matched to primitives of the target platform, e.g., the nextFrame(...) action is marked as asynchronous oneway action. Hence, it becomes a oneway operation in CORBA. Finally, the action's message types are also converted to platform specific elements. Of course, this includes the mapping of complex message types like Image[3]. Note, that the target PIM includes only interaction related elements from the service PIM, i.e., service roles are not considered for the generation of target adaptors.

The target PIM may represent a direct outcome of the development process if the selected target platform requires further processing of interaction role interfaces and types by external tools. For instance, when continuing our case study, the target PIM is used for an M2T transformation to generate *Interface Description Language* (*IDL*) code which is then com-

[1]The mapping of the complex type Image to the Java interface IImage is considered opaque for this example. See Appendix D.2.2 for a more detailed discussion.

[2]Please see Appendix E on page 167 for more details about the prototypical run-time environment used for this case study.

[3]Like for the service PIM/interface model we consider the mapping of complex types opaque.

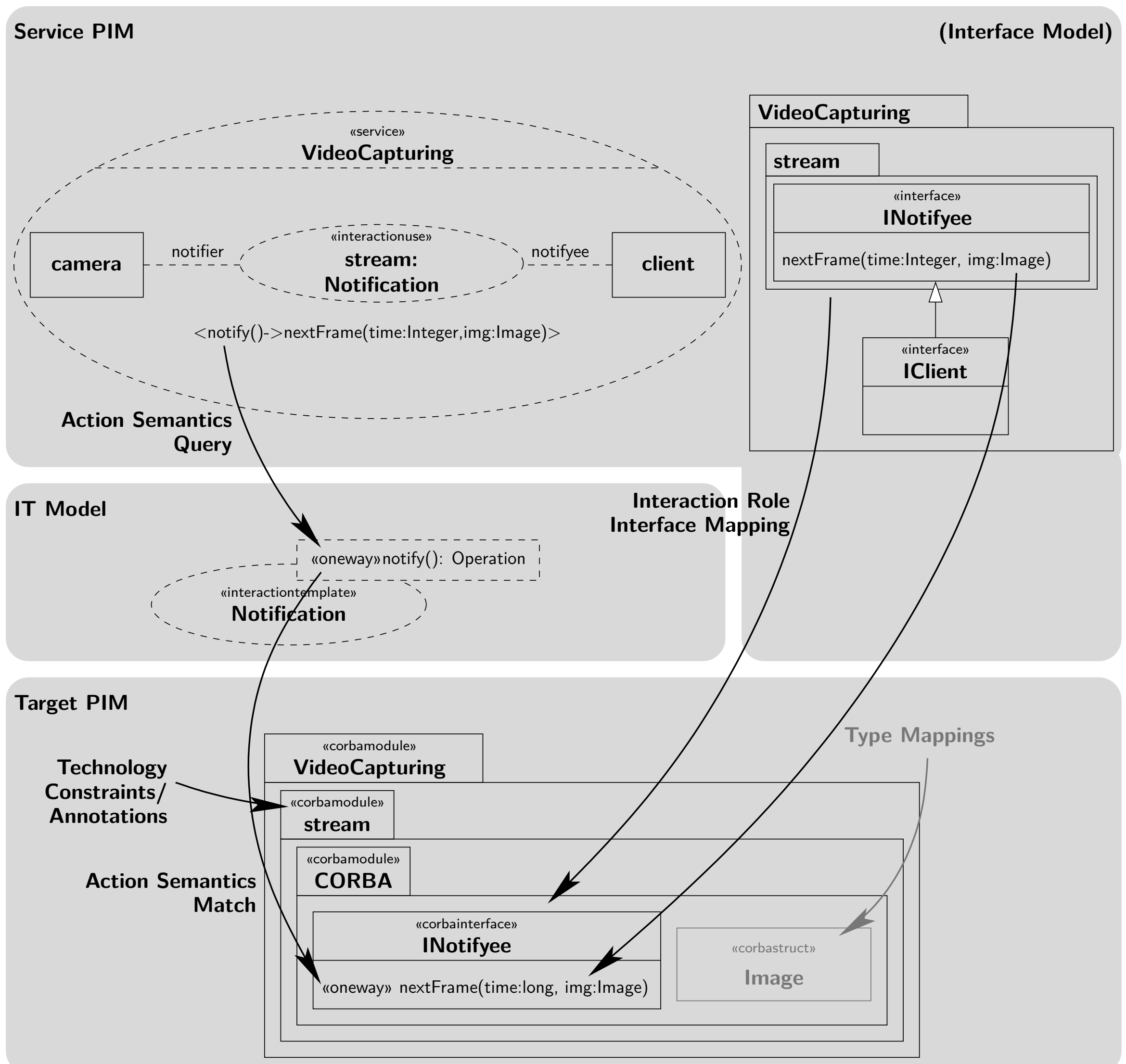


Figure 9.5: Partial target PIM for the video capturing service (reduced to the notifier role of the stream interacting).

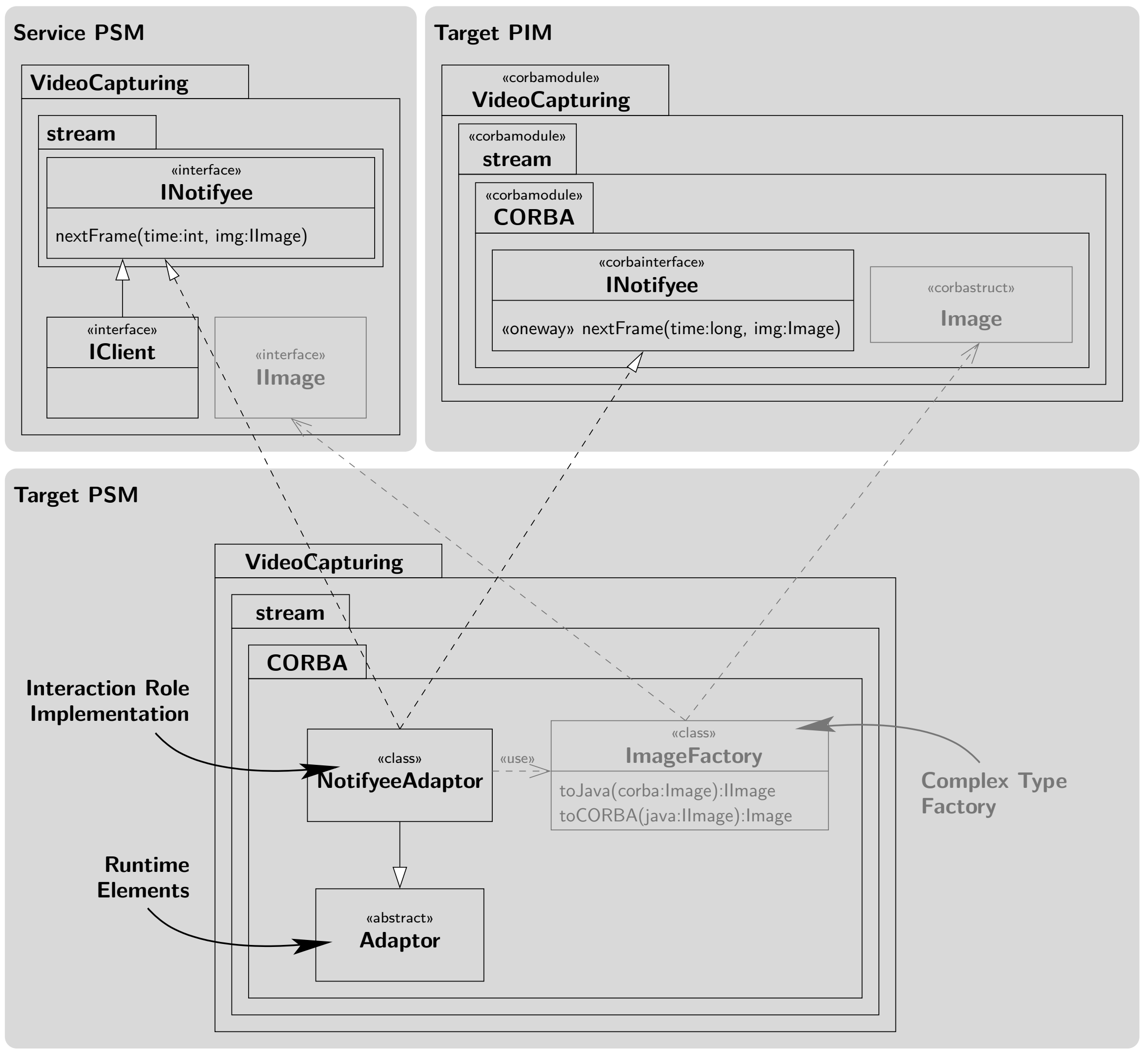


Figure 9.6: Partial target PSM for the video capturing service (simplified and reduced to the notifier role of the stream interacting).

piled by an IDL compiler to create CORBA compliant stub/skeleton code for Java (again, cf. Appendix F on page 171 for the Java/CORBA interfaces resulting from the presented example).

The next step forward to generating deployable target adaptors is to refine the PIM to derive the target PSM. Figure 9.6 shows the target PSM for the PIM in Figure 9.5. The target PSM merges the target technology details of the target PSM with knowledge about how an interaction is mapped to a programming language for service implementation. Hence, it closes the gap between communication technology independent service components and target platform specific realisations. As depicted in Figure 9.6, the NotifyeeAdaptor implements both interaction role interfaces: the one referenced by the service role and the one used for the CORBA mapping. The adaptor mediates between both worlds and is thus responsible for message type conversions, illustrated by the ImageFactory class, and to ensure interaction semantics. The latter refers to the fact, that, like in the given example, the nextFrame(...)

action represents a synchronous operation in Java but is an asynchronous one as defined by the underlying IT. If the underlying middleware, like CORBA, detaches the operation call itself, the adaptor just needs to forward the call. If the middleware can not detach the call, e.g., Java RMI, the adaptor is responsible to realise such behaviour in an opaque manner (cf. Appendix D.1, p. 161). Note, that if the target PIM's interface definitions directly match the ones of the service PIM, i.e., they are totally compatible to each other in terms of action semantics and message types, the target adaptor may be represented by the target PIM itself, solely extended by the run-time elements. That means, that at run-time, a possible derived middleware stub from the target PIM is directly connected to the service role implementation avoiding any further indirection.

Like for a service PSM, the target PSM also contains run-time elements which support the management of target adaptors, e.g., registration within the run-time environment or instantiation for use by service components. Please see Appendix E on page 167 for more details about the prototypical run-time used for this case study.

Finally, we again use an M2T transformation to generate a fully operational implementation of an adaptor as specified by the target PSM. Eventually, this leads to the automated creation of a target adaptor which can be used as a deployment artifact upon system specification.

9.4 Summary

Chapter 9 highlighted the automated parts of the proposed service development process, roughly partitioned into three categories: First, model analysis to determining the service components' required and provided interfaces defined through a service's interactions.Thereby, a component's interface is given by backtracking the component's applied service role, this service role's associated interaction role, and eventually merging the instantiated interfaces of the interaction roles of the underlying ITs.

The second category of automatisation within the process is the generation of skeleton code to ease component implementation. This is achieved by setting the components' interface models into the implementation context, mainly the programming language, by M2M transformations and applying M2T transformations to create source code.

Finally, the third category concerns the creation of target adaptors, i.e., the grounding of the service's interactions to elements of the selected target platform the service components are deployed to. Therefore, the component interface models are enrich with information of how the associated interactions are mapped to primitives of the target platform based on M2M transformations. Again, by also exploiting M2T, eventually individual target adaptors for all the required and provided interfaces of a component are created to facilitate service establishment at run-time.

CHAPTER 10

Conclusions

This chapter summarises the findings of the presented thesis and gives an insight into the prototypical implementation of the proposed service development process. Eventually, potential fields of future work founding on the achieved results are described.

10.1 Results

Based on the increasing complexity of modern avionics, the associated system design processes moved towards MDA based processes [Spi06]. Additionally, the demand for higher system autonomy features [FHS04, Alb03] requires means to further modularise mission systems and to define and establish interactions among the systems' individual components [WFF02]. Therefore, the ideas of service-oriented computing are currently adapted to established, model driven design processes [OWF$^+$07, OFM$^+$07]. Current approaches for service modelling often rely on fixed sets of interaction primitives which in turn rely on features of concrete target platforms, e.g., pure message passing or RPC [KPS06]. In consequence, service modelling is driven by a platform's interaction capabilities instead of being determined by a service's individual communication needs. Hence, on the one hand, service designers are limited in their choice for modelling communication among service participants. On the other hand, the service modelling process strongly focuses on a specific platform, contradicting the principles of MDA.

This thesis presents a novel model-driven design process for interaction-centric services which removes the service designer's limitation on a predefined and closed set of communication primitives as well as the conceptual binding to a specific platform for service specification. The approach strictly separates between aspects of service interaction, the interfaces thereof reflected in application logic, and the realisation of interactions onto target platforms. In detail, the thesis provides the following contributions:

An MDA Development Process. The described interaction-centric service modelling process strongly orients itself on the principles of MDA enabling simplified adaption of the process into established, model based development processes.

A Pattern Language. A pattern language for service interactions was developed supporting the definition of individual ITs as well as libraries thereof to serve as a base for all other modelling activities of the described process.

ITs provide means to represent service interaction semantics by first class modelling entities. An IT captures an interaction pattern in a generic manner, such that it can be applied to varying communication contexts. ITs are collected in an extensible library which is exploited for service definition. Along with its structural and behavioural description, an IT is documented with generic mapping information about how a concrete interaction in the context of a service is to be realised on a specific target platform.

Formal Models. The presented approach supports the formal description of the development process' core concepts, i.e., actions, ITs, interactions, services, and service components.

These formal models serve as a formal grounding to the proposed development process allowing for unambiguous specification of the individual elements and automated validation/verification, to be exploited by an encapsulating system development process.

The UP4IS Profile. Based on the formal grounding of the presented process, a UML profile, referred to as UP4IS, was developed to support graphical IT, interaction, service, and system modelling with standard UML tools.

ITs are modelled via a combination of UML collaboration templates and sequence diagrams to explicitly describe a pattern's communication semantics. Following the interaction-centric idiom, services emerge from the collaboration of their participants. Hence, they are also reflected by UML collaborations, composing individual interactions which are derived from ITs via template instantiation. Finally, components and systems are modelled by component diagrams thereby the individual components are linked to service roles to be realised by the a component's ports.

The Generation of Service Adaptors. Driven by automated model transformations and code generation, the process derives target adaptors to realise service interaction on selected target platforms and assists the service developer in implementing service components.

Automated M2M and M2T transformations are applied to service models to generate stub/skeleton code for interactions, facilitating service implementation. A service's implementation remains independent of the realisation of the underlying interaction itself, directly enabling platform independence of services. Complementing service implementation, a service's interactions have to be realised on concrete target platforms. This process is driven by mappings of the underlying ITs, based on their individual interaction semantics. Using such mappings, groundings of a service's interactions in form of target adaptors are automatically generated. A service's platform independent implementation in combination with the automated realisation of interactions allows for a simplified replacement of target platforms. Additionally, service interactions can be mapped individually to different platforms which enables the combination of complementing technologies for realising distinct interactions within the same service. Thus, peculiarities of alternative platforms can be exploited by selecting a realisation platform based on its support for specific interaction semantics.

Figure 10.1 recapitulates the modelling entities of the proposed development process and their relationships. Following a top down approach, the individual entities represent:

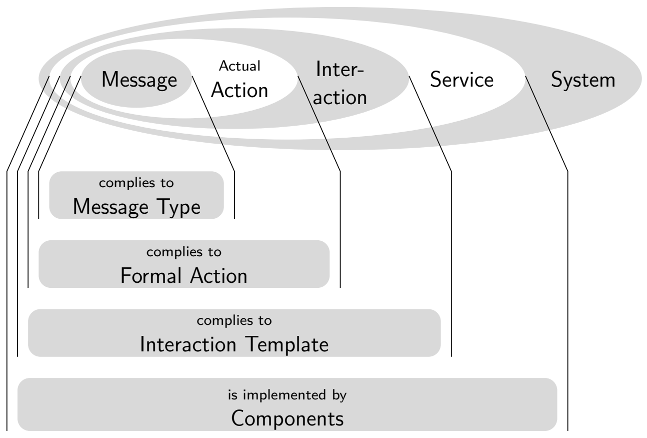


Figure 10.1: The relationship between the proposed modelling concepts.

System. A system is the largest modelled entity of process. A system is a run-time configuration which can be executed to solve a specific task. From a specification point of view, a system is composed out of service components which represent re-usable fragments of collaborative behaviour and the corresponding target adaptors to ground service interactions.

Service. A service is the definition of collaborative functionality, established by the interaction of its participants, referred to as service roles, implemented by components. A service connects its roles through interactions.

Interaction. Interactions describe the communication between a pair of service roles which is necessary to realise the service's collaborative behaviour. Service interactions comply to ITs, applying an IT's interaction pattern to a concrete application context.

Action. An action is the building block for an interaction. It symbolises an atomic block of communication within the specified conversation of an interaction, e.g., passing a message from one interaction participant to another. An IT identifies a number of formal actions which provide means to model generic actions within communication patterns. When instantiating an IT for an interaction definition, the service designer substitutes these formal actions by actual ones and thus puts the pattern into the context of the surrounding service.

Message. A message is used as conceptual container for data transport of actions. An action specifies types of messages, describing which kind of data can be exchanged with an action.

The interaction-centric service development process identifies three actors:

The IT Designer. The IT designer represents the expert who identifies and describes service interaction patterns. He is in charge of maintaining an IT library and contributes generic mapping rules for ITs to concrete target platforms.

The Service Designer. Next, the service designer specifies services by instantiating ITs from the library and integrates them into service definitions to model communication between service participants.

The System Designer. Eventually, the system designer combines service components, i.e., implementations of service roles, with the automatically derived target adaptors for a service's interactions to create run-time configurations.

Although all three roles depend on each other, their individual tasks remain separated by the usage of ITs. For instance, once an IT is modelled, it can already be used by the service designer although the IT designer is still concerned with creating the IT's target mapping specifications.

10.2 Prototype Implementation

The applicability of the proposed development process was demonstrated through the creation of a basic video recording system. Therefore, an IT library was implemented providing eight representative interaction patterns, including notifications, synchronous/asynchronous R/R, and monitorable/abortable derivatives thereof, as well as their target mappings to CORBA, native Java and Java RMI. Based on this library, two services were developed, i.e., the video capturing service and the image compression service. The final video recording system evolves from the composition of both. Based on this case study, the process' model transformations and their relation to the formal models were discussed. The development process was implemented based on the Eclipse[1] framework. The UP4IS profile, the UML service models, the IT library and the necessary M2M as well as M2T transformations were realised based on Eclipse's UML2, OCL2, ATL and M2T frameworks, respectively (cf. Figure 10.2). As prototype deployment platform for the generated services and their target adaptors, the Equinox[2] run-time was used. For more detailed information about individual implementation aspects and generation results refer to Appendices C, D, E, and F.

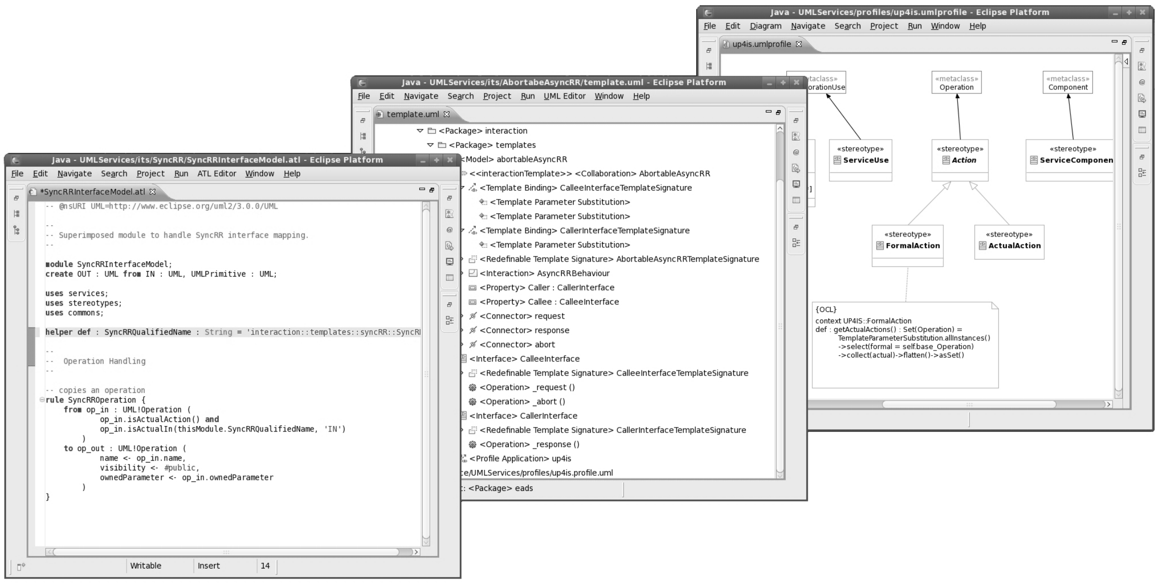
10.3 Future Work

The achievements made in the presented thesis motivate a number of questions to be addressed by future research:

Automatic Target Platform Selection. Every IT possibly comes with a number of target mapping specifications determining how a derived interaction is to be realised on a target platform. In combination with a formal description of a system's run-time environment, the applicable target mappings for a service's interactions can be automatically determined based on the underlying IT's semantics and the middleware support of the deployment platform (cf. [Kur05, GPa$^+$07, TBA04]). Additionally, an interaction can be annotated with *Quality-of-Service* (*QoS*) constraints which would further govern the selection process (cf. [SOA04]).

Derivation of Service Protocols. The behaviour specifications of service interactions, which are inherited from the underlying ITs, could be used to derive complete service protocols. For instance, such a merging process yields the possibilities to validate communication concerning multiple interactions or to generate protocol automata for component

[1]Please refer to `http://www.eclipse.org`.
[2]Please refer to `http://www.eclipse.org/equinox`.


Java - UMLServices/profiles/up4is.umlprofile - Eclipse Platform
File Edit Diagram Navigate Search Project Run Window Help
up4is.umlprofile
«metaclass» OperationUse
«metaclass» Operation
«metaclass» Component
«stereotype» ServiceUse
«stereotype» Action
«stereotype» ServiceComponen
«stereotype» FormalAction
«stereotype» ActualAction
Java - UMLServices/its/AbortabeAsyncRR/template.uml - Eclipse Platform
File Edit Navigate Search Project Run UML Editor Window Help
template.uml
Java - UMLServices/its/SyncRR/SyncRRInterfaceModel.atl - Eclipse Platform
File Edit Navigate Search Project Run ATL Editor Window Help
*SyncRRInterfaceModel.atl
Writable
Insert
14

```
{OCL}
context UP4IS::FormalAction
def : getActualActions() : Set(Operation) =
        TemplateParameterSubstitution.allInstances()
        ->select(formal = self.base_Operation)
        ->collect(actual)->flatten()->asSet()
```

```
<Package> interaction
  <Package> templates
    <Model> abortableAsyncRR
      <<interactionTemplate>> <Collaboration> AbortableAsyncRR
        <Template Binding> CalleeInterfaceTemplateSignature
          <Template Parameter Substitution>
          <Template Parameter Substitution>
        <Template Binding> CallerInterfaceTemplateSignature
          <Template Parameter Substitution>
        <Redefinable Template Signature> AbortableAsyncRRTemplateSignature
        <Interaction> AsyncRRBehaviour
        <Property> Caller : CallerInterface
        <Property> Callee : CalleeInterface
        <Connector> request
        <Connector> response
        <Connector> abort
      <Interface> CalleeInterface
        <Redefinable Template Signature> CalleeInterfaceTemplateSignature
        <Operation> _request ()
        <Operation> _abort ()
      <Interface> CallerInterface
        <Redefinable Template Signature> CallerInterfaceTemplateSignature
        <Operation> _response ()
      <Profile Application> up4is
ce/UMLServices/profiles/up4is.profile.uml
: <Package> eads
```

```
-- @nsURI UML=http://www.eclipse.org/uml2/3.0.0/UML

--
-- Superimposed module to handle SyncRR interface mapping.
--

module SyncRRInterfaceModel;
create OUT : UML from IN : UML, UMLPrimitive : UML;

uses services;
uses stereotypes;
uses commons;

helper def : SyncRRQualifiedName : String = 'interaction::templates::syncRR::SyncRI

--
--  Operation Handling
--

-- copies an operation
rule SyncRROperation {
    from op_in : UML!Operation (
            op_in.isActualAction() and
            op_in.isActualIn(thisModule.SyncRRQualifiedName, 'IN')
        )
    to op_out : UML!Operation (
            name <- op_in.name,
            visibility <- #public,
            ownedParameter <- op_in.ownedParameter
        )
}
```

Figure 10.2: The prototyped modelling tool.

interfaces. The process of merging behavioural specifications would not be fully automated as it requires a service designers knowledge about how individual interactions are inter-weaved (cf. [KCe04, KFJ07, BKB+08]).

Specification of Composite Interaction Patterns. The approach of defining ITs should be extended to allow the specification of *composite* ITs. Composite ITs would be derived from a combination of ITs resulting in a more complex interaction behaviour, but still represent a generic template (cf. [KH06] and [Bir06]). Like for the derivation of service protocols, special attention have to be drawn when mixing the individual models of IT to resolve model conflicts (cf. [CRP08]).

Bibliography

[AAR05] Uwe Aßmann, Mehmet Aksit, and Arend Rensink, editors. *Model Driven Architecture, European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004, Revised Selected Papers*, volume 3599 of *Lecture Notes in Computer Science*, Berlin, Germany, August 2005. Springer.

[ACD$^+$03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Laymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trockovic, and Sanjiva Weerawarana. *Business Process Execution Language for Web Services (BPEL4WS)*, May 2003.

[ADvSP04] João Paulo Almeida, Remco Dijkman, Marten van Sinderen, and Luís Ferreira Pires. Platform-Independent Modelling in MDA: Supporting Abstract Platforms. In Aßmann et al. [AAR05], pages 174–188.

[Alb03] David S. Alberts. Network centric warfare: Curent status and way ahead. *Defence Science*, 8(3), September 2003.

[Alc03] Alcatel, Softeam, Thales, TNI-Valiosys, Codagn Technologies Corp. *Response to the MOF 2.0 Query/View/Transformations RFP*, August 2003.

[Ale79] Christopher Alexander. *The Timeless Way of Building.* Oxford University Press, 1979.

[Alm06] João Paulo Andrade Almeida. *Model-Driven Design of Distributed Applications.* PhD thesis, University of Twente, Enschede, The Netherlands, 2006.

[AO06] Jim Amsden and James Odell. *UML Profile and Metamodel for Services (UPMS) – Request For Proposal.* Object Management Group (OMG), Needham, MA, USA, September 2006.

[AW05] Daniel Amyot and Alan W. Williams, editors. *Proceedings of the 4th International SDL and MSC Workshop (SAM 2004)*, volume 3319 of *Lecture Notes* *in Computer Science*, Ottawa, Canada, January 2005. Springer Berlin/Heidelberg.

[AZ04] Rafik Amir and Amir Zeid. A UML profile for service oriented architectures. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 192–193, New York, NY, USA, 2004. ACM.

[BCD+06] Manfred Broy, Michelle L. Crane, Jürgen Dingel, Alan Hartman, Berhard Rumpe, and Bran Selic. 2nd UML 2 Semantics Symposium: Formal Semantics for UML. In Thomas Kühne, editor, *LNCS4364*, volume 4364 of *Lecture Notes in Computer Science*, pages 318–323, Genoa, Italy, October 2006. Springer Berlin/Heidelberg.

[BCG+05] Boualem Benatallah, Fabio Casati, Daniela Grigori, Hamid R. Motahari Nezhad, and Farouk Toumani. Developing Apadaters for Web Services Integration. In Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen, editors, *LNCS3520*, volume 3520 of *Lecture Notes in Computer Science*, pages 415–429, Porto, Portugal, May 2005. Springer Berlin/Heidelberg.

[BCT04a] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Analysis and Management of Web Service Protocols. In Paolo Atzeni, Wesley Chu, Hongjun Lu, Shuigeng Zhou, and Tok Wang Ling, editors, *LNCS3288*, volume 3288 of *Lecture Notes in Computer Science*, pages 524–541, Shanghai, China, January 2004. Springer Berlin/Heidelberg.

[BCT04b] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Web Service Conversation Modeling: A Cornerstone for E-Business Automation. *IEEE Internet Computing*, 8(1):46–54, 2004.

[BDJ+03] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui. First Experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *Proceedings of the 2003 OOPSLA Workshop on Generative Techniques un the Context of the MDA*, New York, NY, USA, 2003. ACM.

[BdRdSM04] Daniela Berardi, Fabio de Rosa, Luca de Santis, and Massimo Mecella. Finite State Automata As Conceptual Model For E-Services. *Journal of Integrated Design & Process Science*, 8(2):105–121, 2004.

[BDtH05] Alistair Barros, Marlon Dumas, and Arthur H.M. ter Hofstede. Service Interaction Patterns. In *Proceedings of the 3rd International Conference on Business Process Management (BPM 2005)*, volume 3649 of *Lecture Notes in Computer Science*, pages 302–318. Springer, Berlin, Germany, September 2005.

[BF05] Rolv Bræk and Jacqueline Floch. ICT Convergence: Modeling Issues. In Amyot and Williams [AW05], pages 237–256.

[BFGK06] Laura Bocchi, Alessandro Fantechi, László Gönczy, and Nora Koch. Sensoria – D1.1a: Prototype Language for Service Modelling – Ontology for SOAs presented through Structural Natural Language. Technical report, LMU Munich, 2006.

[BGJ99] S. Bremer, M. Glinz, and S. Joos. A Classification of Stereotypes for Object-Oriented Modeling Languages. In *Proceedings of the 2nd International Conference on UML*, pages 249–264, Fort Collins, CO, USA, 1999.

[Bir06] Sebjørn Sæther Birkeland. A Pattern-Based Approach for the Consistent Design of Interaction Interfaces. Master's thesis, Norwegian University of Science and Technology, Department of Telematics, June 2006.

[BKB$^+$08] Olivier Barais, Jacques Klein, Benoit Baudry, Andrew Jackson, and Siobhan Clarke. Composing Multi-View Aspect Models. In *Proceedings of the Seventh IEEE International Conference on Composition-Based Software Systems (ICCBSS 2008)*, pages 43–52, Madrid, Spain, February 2008. IEEE.

[BKM07] Manfred Broy, Ingolf H. Krüger, and Michael Meisinger. A formal model of services. *ACM Transactions on Software Engineering and Methodology*, 16(1):5, 2007.

[BM98] Jean Bezivin and P.-A. Muller. UML: The Birth and Rise of a Standard Modeling Notation. In *First International Workshop on The Unified Modeling Notation UML 98*, pages 1–8. Springer, Berling, Heidelberg, 1998.

[BMB96] Lionel C. Briand, Sandro Morasca, and Victor R. Basili. Property-Based Software Engineering Measurement. *IEEE Transactions of Software Engineering*, 22(1):68–86, 1996.

[BMRS96] Frank Buschmann, Regine Meunier, Hans Rohnert, and Peter Sommerlad. *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley, 1996.

[Boo91] Grady Booch. *Object-Oriented Design with Applications.* Addison-Wesley Book Express, Boston, MA, USA, 1991.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems:* FOCUS *on Streams, Interfaces, and Refinement.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[BS05] Youngjoon Byun and Beverly A. Sanders. A pattern-based development methodology for communication protocols. In *Proceedings of the 2005 ACM symposium on Applied Computing (SAC05)*, pages 1524–1528, New York, NY, USA, 2005. ACM.

[BW05] Lionel Briand and Clay Williams, editors. *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, Berlin, Germany, November 2005. Springer.

[Byu03] Youngjoon Byun. *Pattern-Based Design and Validation of Communication Protocols*. PhD thesis, University of Florida, Gainesville, FL, USA, 2003.

[CB06] H.N. Castejón and R. Bræk. A collaboration-based approach to service specification and detection of implied scenarios. In *Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, page 43. ACM, 2006.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Snajiva Weerawarana. *Web Services Description Language (WSDL) 1.1*. World Wide Web Consortion (WeC), March 2001.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *OOPSLAâĂŹ03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, Anaheim, CA, USA, October 2003.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[Cha04] D. Chappell. *Enterprise service bus*. O'Reilly Media, Inc., 2004.

[COB$^+$08] Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors. *11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008)*, volume 5301 of *Lecture Notes in Computer Science*, Toulouse, France, October 2008. Springer Berlin/Heidelberg.

[CRP08] Antonio Cicchetti, Davide Ruscio, and Alfonso Pierantonio. Managing Model Conflicts in Distributed Development. In Czarnecki et al. [COB$^+$08], pages 311–325.

[DD04] Remco Dijkman and Marlon Dumas. Service-Oriented Design: A Multi-Viewpoint Approach. *International Journal of Cooperative Information Systems (IJCIS)*, 13(4):337–368, December 2004.

[DMRK05] Martin Deubler, Michael Meisinger, Sabine Rittmann, and Ingolf H. Krüger. Modeling Crosscutting Services with UML Sequence Diagrams. In Briand and Williams [BW05], pages 522–536.

[DPW06] Gero Decker, Frank Puhlmann, and Mathias Weske. Formalizing Service Interactions. In *Proceedings of the 4th International Conference on Business Process Management (BPM 2006)*, volume 4102 of *Lecture Notes in Computer Science*, pages 414–419. Springer, Berlin, Germany, October 2006.

[DST04] DSTC, IBM, CBOP. *MOF Query/View/Transformations. Second Revised Submission*, January 2004.

[EK07] Vina Ermagan and Ingolf H. Krüger. A UML2 Profile for Service Modeling. In *LNCS4735*, volume 4735 of *Lecture Notes in Computer Science*, pages 360–374, Nashville, USA, October 2007. Springer Berlin/Heidelberg.

[Erl05] Thomas Erl. *Service-Oriented Architecture – Concepts, Technology, and Design.* Professional Technical Reference. Prentice Hall, New Jersey, USA, 2005.

[Esk99] Philip Eskelin. Component Interaction Patterns. In *Processings of the 6th Conference on Pattern Languages of Programm (PLoP 1999)*, Monticello, IL, USA, August 1999.

[EWA06] Christian Emig, Jochen Weisser, and Sebastian Abeck. Development of soa-based software systems - an evolutionary programming approach. In *Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services*, AICT-ICIW '06, pages 182–, Washington, DC, USA, 2006. IEEE Computer Society.

[Fai98] Ted Faison. Interaction Patterns for Communicating Processes. In *Proceedings of the Pattern Languages of Program Conference*, Monticello, IL, USA, August 1998.

[FHS04] Michael Freed, Robert Harris, and Michael G. Shafto. Humans vs. Autonomous Control of UAV Surveillance. In *1st Intelligent Systems Tech. Conf.*, Chicago, USA, September 2004.

[FLdM96] Kazi Farooqui, Luigi Logrippo, and Jan de Meet. *The ISO Reference Model for Open Distributed Processing - An Introduction.* ISO, February 1996.

[GBPA04] Anastasius Gavras, Mariano Belaunde, Luís Ferreira Pires, and João Paulo A. Almeida. Towards an MDA-Based Development Methodology. In Flávio Oquendo, Brian Warboys, and Ronald Morrison, editors, *Software Architecture*, volume 3047 of *Lecture Notes in Computer Science*, pages 230–240, Berlin, Germany, May 2004. Springer.

[GGKH03] Tracy Gardner, Catherine Griffin, Jana Koehler, and Rainer Hauser. A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard. Technical report, Object Management Group (OMG), July 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[Gil62] Arthur Gill. *Introduction to the Theory of Finite-state Machines.* McGraw-Hill, New York, NY, USA, 1962.

[GMW08] Christian Gierds, Arjan J. Mooij, and Karsten Wolf. Specifying and generating behavioral service adapter based on transformation rules. Technical report, Universität Rostock, 2008.

[GPa$^+$07] Marcela Genero, Mario Piattini, Silvia Mara Abrah ao, Emilio Insfrán, José A. Carsí, and Isidro Ramos. A Controlled Experiment for Selecting Transformations based on Quality Attributes in the context of MDA. In *First International Symposium on Empirical Software Engineering and Measurement, 2007. ESEM 2007.*, page 498, September 2007.

[Hau05] Østein Haugen. Comparing UML 2.0 Interactions and MSC-2000. In Amyot and Williams [AW05], pages 65–79.

[HB05] Dominikus Herzberg and Manfred Broy. Modeling layered distributed communication systems. *Formal Aspects of Computing*, 17(1):1–18, 2005.

[HGM08] Andreas Heil, Martin Gaedke, and Johannes Meinecke. Identifying security aspects in web-based federations. In *IEEE International Conference on Web Services (ICWS 2008)*, pages 807–808, Beijing, China, September 2008.

[HGM09] Andreas Heil, Martin Gaedke, and Johannes Meinecke. Modeling resources in a service-oriented world. In *Hawaii International Conference on System Sciences (HICSS-42)*, pages 1–10, Waikoloa, Big Island, Hawaii, USA, January 2009.

[HLT03] Reiko Heckel, Marc Lohmann, and Sebastion Thöne. Towards a UML Profile for Service-Oriented Architectures. Technical Report TRâĂŞC-TITâĂŞ03âĂŞ27, Faculty of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, Paderborn, Germany, 2003.

[IH05] Stefan Ihmor and Wolfram Hardt. Runtime reconfigurable interfaces - the rtr-ifb approach. *International Journal of Embedded Systems (IJES)*, 1(5/6/2005), 2005.

[ISO04] ISO/IEC. ISO/IEC 15444 series – Information Technology – JPEG 2000 image conding system, 2004.

[JAU06] JAUS. Joint architecture for unmanned systems. `http://www.jauswg.org`, 2006.

[JCC94] Ivar Jacobson, Magnus Christerson, and Larry L. Constantine. The OOSE method: a use – case-driven approach. *Object Development Methods*, pages 247–270, 1994.

[Jen90] Kurt Jensen. Coloured Petri Nets: A High Level Language for System Design and Analysis. In *Advances in Petri Nets*, volume 481 of *Lecture Notes in Computer Science*, pages 342–416. Springer, Berlin/Heidelberg, Germany, 1990.

[Joh05] Simon Johnston. *UML 2.0 Profile for Software Services*. IBM, April 2005.

[KBC04] A. Kalnins, J. Barzdins, and E. Celms. Model transformation language MOLA. In U. Asmann, editor, *Proceedings of Model Driven Architecture: Foundations and Applications 2004*, Linkoping, Sweden, 2004.

[KBW03] Anneke Kleppe, Wim Bast, and Jos B. Warmer. *MDA Explained. The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman, Amsterdam, The Netherlands, 2003.

[KCe04] Jacques Klein, Benoit Caillaud, and Loïc Hélou et. Merging Scenarios. In J. Bicarregui, A. Butterfield, and A. Arenas, editors, *FMICS 2004*, volume 133 of *Electronic Notes in Theoretical Computer Science*, pages 193–215, Linz, Austria, May 2004.

[KFJ07] Jacques Klein, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Multiple Aspects in Sequence Diagrams. In Awais Rashid and Mehment Aksit, editors, *LNCS4620*, volume 4260 of *Lecture Notes in Computer Science*, pages 167–199, Berlin/Heidelberg, Germany, November 2007. Springer Berlin/Heidelberg.

[KH06] Frank Alexander Kraemer and Peter Herrmann. Service Specification by Composition of Collaborations–An Example. In *WI-IATW '06: Proceedings of the 2006 IEEE/WIC/ACM international conference on Web Intelligence and Intelligent Agent Technology*, pages 129–133, Washington, DC, USA, 2006. IEEE Computer Society.

[KKM00] Mati Klip, Ulrich Knauer, and Alexander V. Mikhalev. *Monoids, Acts and Categories with Applications to Wreath Products and Graphs: A Handbook for Students and Researchers*, volume 29 of *De Gruyter Expositions in Mathematics*. Walter de Gruyter, Berlin, Germany, 2000.

[KKRK06] Gerhard Kramler, Elisabeth Kapsammer, Werner Retschitzegger, and Gerti Kappel. Towards Using UML 2 for Modellung Web Service Collaboration Protocols. In Dimitri Konstantas, Jean-Paul Bourrières, Michel Lèonard, and Nacer Boudjlida, editors, *IESA*, pages 227–238. Springer London, London, UK, July 2006.

[KM04] Ingolf H. Krüger and Reena Mathew. Systematic Development and Exploration of Service-Oriented Software Architectures. In *WICSA 2004*, pages 177–187, Washington, DC, USA, June 2004. IEEE Computer Society.

[KM08] Pierre Kelsen and Qin Ma. A Lightweight Approach for Defining the Formal Semantics of a Modeling Language. In Czarnecki et al. [COB+08], pages 690–704.

[KMH+07] Nora Koch, Philip Mayer, Reiko Heckel, László Gönczy, and Carlo Montangero. Sensoria – D1.4a: UML for Service-Oriented Systems. Technical report, LMU Munich, Munich, Germany, October 2007.

[KPS06] Raman Kazhamiakin, Marco Pistore, and Luca Santuari. Analysis of communication models in web service compositions. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 267–276, New York, NY, USA, 2006. ACM.

[Krü04] Ingolf H. Krüger. Service Specification with MSCs and Roles. In *Proceedings of IASTED International Conference on Software Engineering*, Insbruck, Austria, 2004.

[Kur05] Ivan Kurtev. *Adaptability of Model Transformations*. PhD thesis, University of Twente, Enschede, The Netherlands, 2005.

[KvdB03] Ivan Kurtev and Klaas van den Berg. A Synthesis-Based Approach to Transformations in an MDA Software Development Process. In Arend Rensink, editor, *MDAFA2003*, pages 121–126, Enschede, The Netherlands, June 2003. University of Twente.

[Lar02] Craig Larman. *Applying UML and Patterns.* Prentice-Hall, Englewood Cliffs, NJ, USA, 2 edition, 2002.

[LFW$^+$08] Xitong Li, Yushun Fan, Jian Wang, Li Wang, and Feng Jiang. A Pattern-Based Approach to Development of Service Mediators for Protocol Mediation. In *WICSA '08: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, pages 137–146, Washington, DC, USA, 2008. IEEE Computer Society.

[LGW$^+$07] Sergey Likuchev, Adrian Giurca, Gerd Wagner, Dragan Gasevic, and marko Ribaric. Using UML-based Rules for Web Services Modeling. In *23rd International Conference on Data Engineering*, pages 290–297, Washington, DC, USA, 2007. IEEE Computer Society.

[Lin06] Niklas Lindén. Middleware Technologies For Online Games - MoM or RPC. Blekinge Institute of Technology Student Workshop on Architectures and Research in Middleware (BITSWARM2006), January 2006.

[LJJ05] Zheng Li, Jun Jan, and Yan Jin. Pattern-Based Specification and Validation of Web Services Interaction Properties. In Boualem Benatallah, Fabio Casati, and Paolo Traverso, editors, *LNCS3826*, volume 3826 of *Lecture Notes in Computer Science*, pages 73–86, Amsterdam, The Netherlands, November 2005. Springer Berlin/Heidelberg.

[LPS03] Sea Ling, Iman Poernomo, and Heinz Schmidt. Describing Web Service Architectures through Design-by-Contract. In *Proceedings of the 18th International Symposium on Computer and Information Sciences (ISCIS 2003)*, volume 2869 of *Lecture Notes in Computer Science*, pages 1008–1018, Antalya, Turkey, October 2003. Springer Berlin, Heidelberg.

[LSACM08a] Marcos López-Sanz, César J. Acuna, Carlos E. Cuesta, and Esperanza Marcos. Defining Service-Oriented Software Architecture Models for a MDA-based Development Process at the PIM level. In *WICSA '08: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, pages 309–312, Washington, DC, USA, 2008. IEEE Computer Society.

[LSACM08b] Marcos López-Sanz, César J. Acuòa, Carlos E. Cuesta, and Esperanza Marcos. Modelling of Service-Oriented Architectures with UML. *Electronic Notes in Theoetical Computer Science (ENTCS)*, 194(4):23–37, 2008.

[MBLN06] Ayman Mahfouz, Leonor Barroca, Robin Laney, and Bashar Nuseibeh. Patterns for service-oriented information exchange requirements. In *PLoP '06: Proceedings of the 2006 conference on Pattern languages of programs*, pages 1–10, New York, NY, USA, 2006. ACM.

[Mei09] Johannes Meinecke. *Supporting the Evolution of Federated Systems in Web Engineering.* PhD thesis, Chemnitz University of Technology, Chemnitz, Germany, May 2009.

[Men07] Falko Menge. Enterprise Service Bus. In *Proceedings of the Free and Open Source Software Conference 2007 (FrOSCon 2007)*, August 2007.

[MLM+06] C. Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F Brown, and Rebekah Metz. *Reference Model for Service Oriented Architecture 1.0.* OASIS, August 2006.

[MM03] Jishnu Mukerji and Joaquin Miller. *Modell Driven Architecture (MDA) Guide Version 1.0.1.* Object Management Group (OMG), Needham, MA, USA, June 2003.

[MORF09] Herwig Moser, Norbert Oswald, Toni Reichelt, and Stefan Förster. Effective Information Management in Joint Operations based on Semantic Technologies. In *Proceedings of the NATO ROT IST-087 Symposium on Information Management/Exploitation*, Stockholm, Sweden, 2009. NATO RTO.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–40, September 1992.

[MROF08] Herwig Moser, Toni Reichelt, Norbert Oswald, and Stefan Förster. Information Management for Unmanned Systems: Combining DL-Reasoning with Publish/Subscribe. In *Proceedings of SGAI 2008*, Cambridge, UK, 2008. Springer.

[MROF09a] Herwig Moser, Toni Reichelt, Norbert Oswald, and Stefan Förster. Context-sensitive Plan Execution Language for Adaptive Robot Behaviour. In *Proceedings of 29th SGAI International Conference on Artificial Intelligence*, pages 233–246, Cambridge, UK, 2009. Springer.

[MROF09b] Herwig Moser, Toni Reichelt, Norbert Oswald, and Stefan Förster. PLEXIL-DL: Language and Runtime for Context-Aware Robot Behaviour. In *Proceedings of FIRA 2009*, pages 179–186, Incheon, Korea, 2009.

[NRB+07] M. Nezhad, H. Reza, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *Proceedings of the 16th International Conference on the World Wide Web*, page 1002. ACM, 2007.

[NTER06] Jin Nakazawa, Hideyuki Tokuda, W. Keith Edwards, and Umakishore Ramachandran. A Bridging Framework for Universal Interoperability in Pervasive Systems. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 3, Washington, DC, USA, 2006. IEEE Computer Society.

[Ö06] Turhan Özgür. The Middleware Challenges for Software Engineering. Blekinge Institute of Technology Student Workshop on Architectures and Research in Middleware (BITSWARM2006), January 2006.

[OAS07] OASIS. *Business Process Execution Language (BPEL).* OASIS, April 2007.

[OFM+07] Norbert Oswald, Stefan Förster, Herwig Moser, Toni Reichelt, and André Windisch. An Architectural Framework for Cooperative Civil and Military Mission Scenarios. In Karsten Berns and Tobias Luksch, editors, *Autonome Mobile Systeme 2007*, pages 110–113. Springer, 2007.

[oI01] ITU-T Telecommunication Standardization Sector of ITU. *ITU-T Recommendation Z.120 – Message Sequence Chart (MSC).* International Telecommunication Union (ITU), 2001.

[OMG01] OMG. *Model Driven Architecture (MDA).* Object Management Group (OMG), Needham, MA, USA, July 2001.

[OMG02a] OMG. MOF 2.0 Query/View/Transformations Request for Proposal. Technical report, Object Management Group (OMG), October 2002.

[OMG02b] OMG. *UML Profile for CORBA Specification.* Object Management Group (OMG), Needham, MA, USA, April 2002.

[OMG04] OMG. *Meatmodel and UML Profile for Java and EJB Specification.* Object Management Group (OMG), Needham, MA, USA, February 2004.

[OMG05a] OMG. *A Proposal for an MDA Foundation Model.* Object Management Group (OMG), Needham, MA, USA, April 2005.

[OMG05b] OMG. *MOF QVT Final Adopted Specification.* Object Management Group (OMG), Needham, MA, USA, November 2005.

[OMG06] OMG. *Meta Object Facility (MOF) Core Specification v2.0.* Object Management Group (OMG), Needham, MA, USA, January 2006.

[OMG07a] OMG. *MOF Models to Text Transformation Language.* Object Management Group (OMG), Needham, MA, USA, August 2007.

[OMG07b] OMG. *OMG Unified Modeling Language (OMG UML) – Object Constraint Language (OCL) v2.1.2.* Object Management Group (OMG), Needham, MA, USA, November 2007.

[OMG08a] OMG. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1 – CORBA Interfaces.* Object Management Group (OMG), Needham, MA, USA, January 2008.

[OMG08b] OMG. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1 – CORBA Interoperability.* Object Management Group (OMG), Needham, MA, USA, January 2008.

[OMG09a] OMG. *OMG Unified Modeling Language (OMG UML) – Infrastructure v2.2.* Object Management Group (OMG), Needham, MA, USA, February 2009.

[OMG09b] OMG. *OMG Unified Modeling Language (OMG UML) – Superstructure v2.2.* Object Management Group (OMG), Needham, MA, USA, February 2009.

[OMG11] OMG. *Business Process Model and Notation (BPMN).* Object Management Group (OMG), Needham, MA, USA, January 2011.

[OSG11] OSGi Alliance. *OSGi Service Platform Core Specification – Release 4, Version 4.3*, April 2011.

[OWF$^+$07] Norbert Oswald, André Windisch, Stefan Förster, Herwig Moser, and Toni Reichelt. A Service-oriented Framework for Manned and Unmanned Systems to support Network-centric Operations. In *Proceedings of the Fourth International Conference on Informatics in Control, Automation and Robotics*, pages 284–291, Angers, France, May 2007. Institute for Systems and Technologies of Information, Control and Communication (INSTICC), INSTICC Press.

[Pap03] Mike P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering*, pages 3–12, Washington, DC, USA, 2003. IEEE Computer Society.

[Pat04] O. Patrascoiu. YATL: Yet Another Transformation Language. In M. van Sinderen and L. Pires, editors, *Proceedings of the 1st European MDA Workshop on Industrial Applications (MDA-IA)*, Enschede, The Netherlands, 2004.

[PB03] Rachel A. Pottinger and Philip A. Bernstein. Merging models based on given correspondences. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 862–873. VLDB Endowment, 2003.

[PCS07] Linh Duy Pham, Alan Colman, and Jean-Guy Schneider. Dynamic Protocol Aggregation and Adaptation for Service-Oriented Computing. In *ASWEC '07: Proceedings of the 2007 Australian Software Engineering Conference*, pages 39–48, Washington, DC, USA, 2007. IEEE Computer Society.

[Pet81] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[PRFS07] Mikhail Perepletchikov, Caspar Ryan, Keith Frampton, and Heinz W. Schmidt. A Formal Model of Service-Oriented Design Structure. In *Proceedings of the 18th Australian Conference on Software Engineering (ASWEC 2007*, pages 71–80, Melbourne, Victoria, Australia, April 2007.

[QVT03] QVT-Partners. *Revised submission for MOF 2.0 Query/View/Transformations RFP*, 2003.

[RB95] James R. Rumbaugh and Grady Booch. *Unified Method.* Rational Software Corporation, Santa Clara, CA, USA, 1995.

[RBL$^+$90] James R. Rumbaugh, Michael R. Blaha, William Lorensen, Frederick Eddy, and William Premerlani. *Object-Oriented Modeling and Design.* Prentice-Hall, Upper Saddle River, NJ, USA, 1990.

[RF03] Pablo Rossi and George Fernandez. Definition and Validation of Design Metrics for Distributed Applications. In *Proceedings of the 9th International Symposium on Software Metrics (METRICS '03)*, pages 124–133, Washington, DC, USA, 2003. IEEE Computer Society.

[ROW$^+$07] Toni Reichelt, Norbert Oswald, André Windisch, Stefan Förster, and Herwig Moser. IP Based Transport Abstraction for Middleware Technologies. In

Proceedings of the 3rd International Conference on Networking and Services, pages 39–43, Athens, Greece, 2007. IEEE Computer Society.

[RQZ07] Chris Rupp, Stefan Queins, and Barbara Zengler. *UML 2 glasklar*. Carl Hanser Verlag, Munich, Germany, 3 edition, 2007.

[San07] Richard Torbjørn Sanders. *Collaboration, Semantic Interfaces and Service Goals: a way forward to Service Engineering*. PhD thesis, Norwegian University of Science and Technology, Trondheim, Norway, March 2007.

[SHLP05] M.T. Schmidt, B. Hutchison, P. Lambros, and R. Phippen. The enterprise service bus: Making service-oriented architecture real. *IBM Systems Journal*, 44(4):781–797, 2005.

[SK05] Miroslaw Staron and Ludwik Kuzniarz. Properties of Stereotypes from the Perpective of Their Role in Designs. In Briand and Williams [BW05], pages 201–216.

[SK06] Miroslaw Staron and Ludwik Kuzniarz. Transformational Stereotypes: A Support for Transforming UML models. In *Nordic Workshop on UML*, Grimstadt, Norway, 2006. Blekinge Institute of Technology.

[SOA04] Arnor Solberg, Jon Oldevik, and Jan Øyvind Aagedal. A Framework for QoS-Aware Model Transformation, Using a Pattern-Based Approach. In Robert Meersman and Zahir Tari, editors, *OTM Confederated International Conferences 2004*, volume 3291 of *Lecture Notes in Computer Science*, pages 1190–1207. Springer, Berlin, Germany, October 2004.

[Spi00] Cary R. Spitzer. *Digital Avionics Systems*. The Blackburn Press, Caldwell, NJ, USA, 2000.

[Spi06] Cary R. Spitzer. *Avionics: Development and Implementation*. CRC Press, Boca Raton, FL, USA, 2006.

[TBA04] Bedir Tekinerdogan, Sevcan Bilir, and Cem Abatlevi. Integrating Platform Selection Rules in the Model Driven Architecture Approach. In Aßmann et al. [AAR05], pages 159–173.

[TvS01] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall International, 2001.

[V07] Markus Völter. Service, Komponenten, Modelle. In Gernot Starke and Stefan Tilkov, editors, *SOA-Expertenwissen*, chapter 25, pages 423–438. dpunkt.verlag GmbH, 1st edition, 2007.

[VCH10] Matthias Vodel, Mirko Casper, and Wolfram Hardt. Embedded Ambient Networking - A New, Lightweight Communication Concept. In *Proceedings of the 2010 International Conference on Communications (ICC2010)*, Cape Town, South Africa, May 2010.

[VCM05] Juan M. Vara, Valeria De Castro, and Esperanza Marcos. WSDL Automatic Generation from UML MOdels in a MDA Framework. *International Journal of Web Services Practices*, 1(1-2):1–12, 2005.

[vdBC01] Just A. van den Broecke and James O. Coplien. Using design patterns to build a framework for multimedia networking. In Linda Rising, editor, *Design patterns in communications software*, pages 259–292, New York, NY, USA, 2001. Cambridge University Press.

[VE10] Matthias Vodel and Wolfram Hardt (Ed.). *Funkstandardübergreifende Kommunikation in Mobilen Ad Hoc Netzwerken*, volume 9 of *Wissenschaftliche Schriftenreihe Eingebettete Selbstorganisierende Systeme.* Universitätsverlag Chemnitz, Juli 2010.

[WFF02] André Windisch, Marco Fischer, and Stefan Förster. A re-use orientated design methodology for future integrated modular avionics systems. In *FAST'02*, pages 147–153, London, UK, October 2002.

[Wil03] E. Willink. UMLX: A graphical transformation language for MDA. In A Rensink, editor, *Proceedings of Model Driven Architecture: Foundations and Applications 2003*, 2003.

[YZCM04] Z. Yang, Z. Zhou, B.H.C. Cheng, and P.K. McKinley. Enabling collaborative adaptation across legacy components. In *ARM '04: Proceedings of the 3rd workshop on Adaptive and reflective middleware*, pages 277–282, New York, NY, USA, 2004. ACM.

[ZCCK04] Jia Zhang, Jen-Yao Chung, Carl K. Chang., and Seong W. Kim. WS-Net: A Petri-net Based Specification Model for Web Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS '04)*, pages 420–427, Washington, DC, USA, 2004. IEEE Computer Society.

[ZDtH$^+$06] Johannes Maria Zaha, Marlon Dumas, Arthur ter Hofstede, Alistair Barros, and Gero Decker. Service Interaction Modeling: Bridging Global and Local Views. In *EDOC '06: Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference*, pages 45–55, Washington, DC, USA, 2006. IEEE Computer Society.

[Zim80] Hubert Zimmermann. OSI Reference Model – The OSI Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communication*, 28(4):425–432, April 1980.

Part IV

Appendices

APPENDIX A

UML Foundations

A.1 Introduction

The *Unified Modeling Language* (*UML*) is an standardised, object-oriented modelling language providing system architects and software engineers tools for analysing, designing and implementing software based systems or business processes [OMG09b]. Like MDA, UML is also hosted by the OMG. The UML is designed to be a human readable, graphical annotation language based on a modelling approach compatible to MDA, representing a *Meta-Model* (*M2*) language. It is important to note, that the UML is neither a formal language with well defined semantics[1] nor represents a complete system design methodology. The power of this language lies within it's concrete application to a given problem as well as the use of appropriate design tools.

The development of the UML was motivated by the demand for a unified notation of object-oriented systems to enable information exchange between cooperating companies as well as the academic world. In the early 1990s, there existed various notations and dialects being used for system modelling. The most prominent examples are the *Object Modelling Technique* (*OMT*) by Rumbaugh [RBL+90], the *Object-Oriented Design* (*OOD*) by Booch [Boo91], and the *Object-Oriented Software Engineering* (*OOSE*) approach of Jacobson [JCC94]. After Rumbaugh and Booch had already merged their approaches to the *Unified Method* in 1995 [RB95], Jacobson joined both to present a first draft of UML (version 0.9) in 1996 and the final version 1.0 one year later (cf. Figure A.1). They addressed four goals (cf. [BM98]):

- provide notation concepts supporting several, object-oriented views on the same system,
- draw a strong relation between the concepts and the implementation of a system,
- address system maintenance and scalability, and
- allow both, human and machine readability.

[1]However, there is ongoing research to define formal semantics for current version of UML (2.0), cf. [BCD+06] and [KM08].

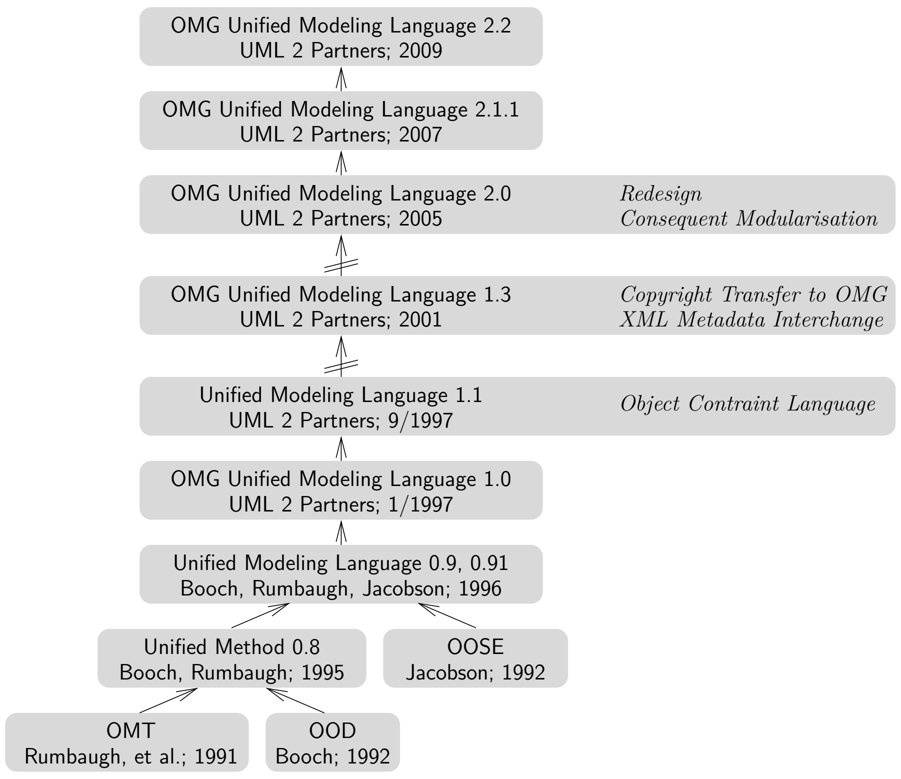

OMG Unified Modeling Language 2.2
UML 2 Partners; 2009
OMG Unified Modeling Language 2.1.1
UML 2 Partners; 2007
OMG Unified Modeling Language 2.0
UML 2 Partners; 2005
Redesign
Consequent Modularisation
OMG Unified Modeling Language 1.3
UML 2 Partners; 2001
Copyright Transfer to OMG
XML Metadata Interchange
Unified Modeling Language 1.1
UML 2 Partners; 9/1997
Object Contraint Language
OMG Unified Modeling Language 1.0
UML 2 Partners; 1/1997
Unified Modeling Language 0.9, 0.91
Booch, Rumbaugh, Jacobson; 1996
Unified Method 0.8
Booch, Rumbaugh; 1995
OOSE
Jacobson; 1992
OMT
Rumbaugh, et al.; 1991
OOD
Booch; 1992

Figure A.1: The UML and it's history (based on [RQZ07, p. 14]).

The next major milestones were the integration of the OCL (UML 1.1) , the transfer of copyrights to the OMG and definition of a textual representation in form of the *XML Metadata Interchange Format* (*XMI*) (UML 1.3), and finally the release of major version 2.0 (current minor version 2.2 [OMG09b, OMG09a]), containing an overall redesign and closer alignment to related OMG standards, e.g., OCL and MOF [OMG09a, p. 14, pp. 209–210].

A.2 The UML Version 2

With the advancement of UML 1.x, the UML was extended by more and more features and partially competing concepts. Hence, the primary goals for the specification of UML 2 were an overall cleanup with the elimination of redundant modelling concepts and clear separation of them, the clarification of diagram semantics, and a tighter integration and consequent use of other OMG standards, e.g., the exclusive application of OCL to express diagram constraints. Additional goals addressed specific trends in software design: the integration of a notion of time for behaviour specifications, thus enabling better support for real-time applications, and simplifying component oriented system design.

The current specification of UML (version 2.2 from 2009) is split into two parts. The first part represents the UML *infrastructure* [OMG09a], describing fundamental language constructs and their architectural relation among each other. Based on this document, the second part, the UML *superstructure* [OMG09b] defines available diagram notations in conjunction with their semantics. This bi-partitioning leads to a cleaner and more compact language specification with increased re-use of language concepts. However, the price of the

UML 2 redesign is the loss of compatibility to version 1.x, e.g., the elimination or realignment of diagrams. Furthermore, UML 2 still lacks complete, enforcing semantics. Beside the possibility of so called *presentation options*, which allow alternative notations of the same concepts, the language still contains semantic gaps, referred to as *semantic variation points*. For these variation points, UML semantics are either not defined at all, or several options about the meaning and interpretation of effected concepts are presented.

A.3 The Notion of Model in UML

The notion of *model* in the UML is given as capturing "a view of a physical system. It is an abstraction of the physical system, with a certain purpose." [OMG09b, p. 615]. The purpose determines what is to be covered by the model and what is left out, being irrelevant to the model's context. A model may contain elements of four major categories: *types*, *classifiers*, *events*, and *behaviours* [OMG09b, pp. 13–14]. Types, also referred to as data types, describe model elements which instances are identified only by their values. A classifier describes individuals, referred to as *objects*, of a system having a state and being set into relation to other individuals. An event describes *occurrences*, i.e., something that happens and which occurrence will have consequences on the modelled system. Finally, a behaviour describes a set of *executions*. Hereby, an execution is defined by a set of rules specifying an algorithm. It is important to emphasise the difference between objects and models of objects. In UML, models do not directly contain objects, but objects are subject of models. Hence, models describe objects, occurrences, and executions with similar properties by abstraction.

A.4 The UML Meta-Model

The UML meta model was designed with the following main goals in mind [OMG09a, p. 11]: *modularity*, *layering*, *partitioning*, *extensibility*, and *reuse*. Modularity of the language specification is based on the idea of grouping different constructs into packages, i.e., closed namespaces, and organising features into meta classes. Layering is addressed in two ways. First, the UML core constructs and thereof derived higher-level constructs are separated from each other using the package mechanism, and second, the overall modelling process strongly follows the four layer principle (see Section 3.2.1). Partitioning is supported by the UML to allow conceptual organisation of constructs of of the same layer, supporting further separation of concerns. In that sense, UML not only offers a fine grained meta-model library, being consequently reused for the language specification itself, but also integrates other related meta-models like MOF. Lastly, extensibility is a major advantage of UML, making the language adaptable to domain specific needs. Extensibility is discussed in more detail in Section 4.6.

The UML 2 language specification is decomposed in two orthogonal ways. First, the language is split vertically into so called *language units* [OMG09a, p. 2], concerning different facets of the language. Language units group closely related concepts of the specification together in just one part, e.g., the language part being used for state machine diagrams forms one independent language unit. Some language units are even more partitioned into multiple increments allowing for a finer grained application or usage of specific UML concepts. The language unit principle results in two advantages for UML specification. First, the individual units may be advanced independently of each other. Second, users, as well as tool designers, can select and focus only on the UML features being relevant for their work.

The second decomposition of UML is provided horizontally, i.e., the layered design of UML language concepts. Hereby, lower layers contain simpler constructs of language units. To ensure modelling and tool compatibility, the OMG has defined the *compliance levels* 0, 1, 2, and 3 [OMG09b, p. 8–9]. A tool supporting compliance level 0 does only support the core concepts of UML, e.g., Class or Package. In contrast, compliance level 3 comprises the complete UML language specification including, but not limited to, extended association classes, collaborations, and templates.

A.5 Diagram Types

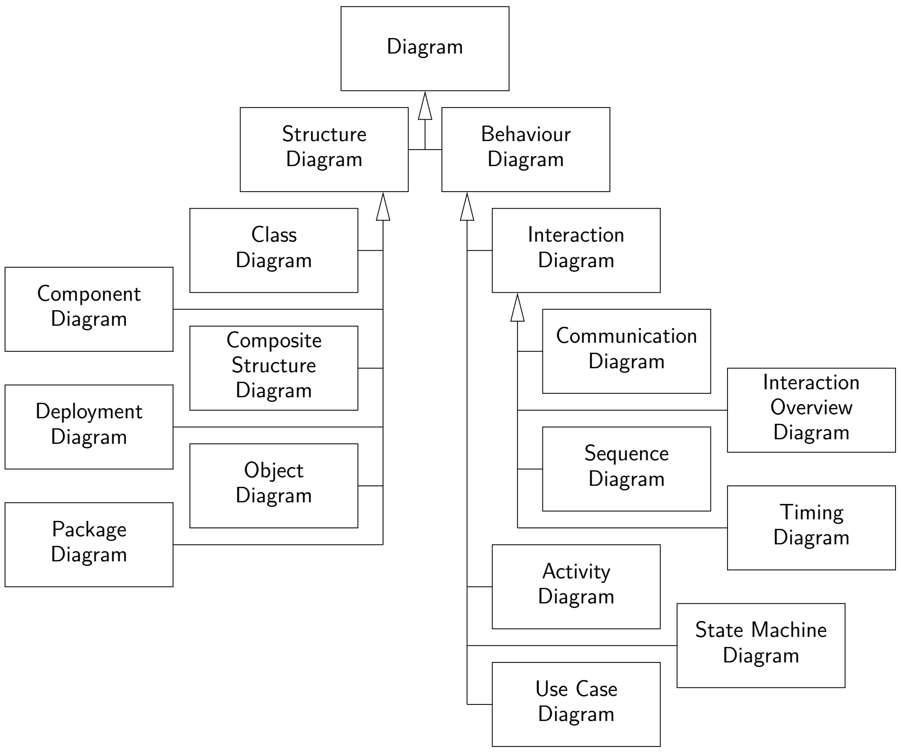


Figure A.2: The UML diagram types.

UML 2 defines 13 diagram types [OMG09b], being split into structural diagrams on the one hand and behavioural diagrams on the other hand (cf. Figure A.2). The first part of the UML superstructure specification concerns the *Structure*. It contains diagrams to be used to describe the static, structural entities of a system:

Class Diagram. A class diagram reflects the detailed, object-oriented structural design of a system. It describes individual classes as type definition of object instances. Classes contain *attributes* and *operations*. Furthermore, a class diagram may express relations between classes through *associations*.

Component Diagram. A component diagram visualises parts of the fundamental run-time structure or architecture of the system under design. The main focus lies on *components* as modular, replaceable parts of a system. Therefore, component diagrams support the specification of the closure of a component, i.e., its interaction with the environment through required and provided *interfaces*.

Composite Structure Diagram. A composite structure diagram shows the internal structures of a classifier, e.g., a class or component, and their relations amongst each other. A special variant of this diagram is the *collaboration diagram* showing the different *roles* and interplay of system components.

Deployment Diagram. A deployment diagram describes the run-time distribution of software components to hardware units of the system.

Object Diagram. An object diagram exemplifies the run-time configuration of a system. The diagrams represents a snapshot of a running system showing the concrete object instances and their currently established relations. An object diagram can be seen as a, possibly partial, instantiation of a class diagram.

Package Diagram. A package diagram groups aspects of a system model together in one package, i.e., different views on a system. Packages are used for logical structuring of a system design.

Dynamic aspects, or *Behaviour*, of systems are addressed by the second part of the UML superstructure specification through behaviour diagrams.

Communication Diagram. A communication diagram shows interactions between communicating entities in an abstract manner. The focus lies on the interplay of different interactions and how interactions are initiated.

Sequence Diagram. A sequence diagram explains the communication between parts or roles of a system. Based on the concept of message sequences, sequence diagrams are closely related to MSCs [oI01] (cf. Section 4.4.3).

Interaction Overview Diagram. An interaction overview diagram correlates individual interactions by presenting a more abstract view on the interactions occurring in a system. The diagram focuses on the causal ordering of individual interactions. Interaction overview diagrams can be understood as compositions of sequence diagrams where each diagram is used as a black box element.

Timing Diagram. A timing diagram shows the state changes of a system, or subsystem, over time, based on a concrete time scale.

Activity Diagram. An activity diagrams describes work-flows of a system, e.g., by modelling the behaviour of an operation. An activity diagrams shows how a behaviour is realised.

State Machine Diagram. A state machine diagram allows the modelling of hierarchical finite state machines.

Use Cases. A use case diagram reflects the analysis of system requirements. A use case describes a closed application scenario of a system by putting users and system actions in relation to represent the fulfilment of a specific task of the system.

APPENDIX B

Mathematical Foundations

B.1 Tuples

Given some non-empty set X, a tuple is defined as a finite sequence of elements from X. X is referred to as *domain* of the tuple's elements A tuple of n elements is generally referred to as *n-tuple*. For small n, it is common to use special names for tuples, e.g., *pair* ($n = 2$), or *triple* ($n = 3$). For a tuple consisting of the elements $(x_1, \ldots, x_n)$, we write x_i with $1 \leq i \leq n$ to refer to the i-th element of the tuple.

An *homogeneous* n-tuple T over elements from set X is an ordered sequence of n elements from X. It is denoted as $(x_1, \ldots, x_n) \in X^n$. The cardinality, or length, of an n-tuple is $|T| = n$. A homogeneous tuple $T = (x_1, \ldots, x_n)$ with *pair-wise disjoint* elements is a tuple which does not contain the same element twice at different positions, i.e.,

$$\underset{1 \leq i,j \leq n}{\forall} x_i = x_j \Rightarrow i = j$$

In contrast to homogeneous tuples, an *heterogeneous* n-tuple T is a ordered sequence of elements from different domains, i.e., $T = (x_1, \ldots, x_n) \in X_1 \times \ldots \times X_n$.

B.2 Binary Relations

Given two sets A and B, a binary relation R is defined as a subset of the Cartesian product of these sets, i.e., $R \subseteq A \times B$. Thus, R is a set of pairs whose first component is an element of A and second component is an element of B, respectively. Based on [KKM00, p. 3], we define some properties of binary relations. We say R is *left-unique*, if every element of B is in relation to at most one element of A, i.e.:

$$\underset{b \in B}{\forall} \underset{a_1, a_2 \in A}{\forall} (a_1, b) \in R \land (a_2, b) \in R \Rightarrow (a_1 = a_2)$$

R is *right-unique*, if every every element of A is in relation to at most one element of B, i.e.:

$$\underset{a \in A}{\forall} \underset{b_1, b_2 \in B}{\forall} (a, b_1) \in R \land (a, b_2) \in R \Rightarrow (b_1 = b_2)$$

R is called *left-total*, if all elements of A are related to at least one element of B, i.e.:

$$\underset{a \in A}{\forall} \; \underset{b \in B}{\exists} : (a, b) \in R$$

R is called *right-total*, if all elements of B are related to at least one element of A, i.e.:

$$\underset{b \in B}{\forall} \; \underset{a \in A}{\exists} : (a, b) \in R$$

The complement $\overline{R}$ of relation R is the set of pairs relating elements of A to elements of B which are not in relation to each other within R, i.e.:

$$\overline{R} = \{(a, b) | a \in A \wedge b \in B \wedge (a, b) \notin R\}$$

APPENDIX C

OCL Formalisation of the UP4IS Stereotype Constraints

This chapter lists the OCL formalisations of the stereotype constraints in the UP4IS UML profile presented in Chapter 6.

C.1 Action

1. An action has public visibility as an action provides means for external communication of a component.

   ```
   context UP4IS::Action
   inv: self.base_Operation.visibility = #public
   ```

2. An action's parameters have the direction in, out, or return.

   ```
   context UP4IS::Action
   inv: self.base_Operation.ownedParameter
          ->forAll(p | p.direction = #in or
                       p.direction = #out or
                       p.direction = #return)
   ```

C.2 ActualAction

No additional constraints.

C.3 FormalAction

1. A formal action defines no parameters or exceptions.

   ```
   context UP4IS::FormalAction
   inv: self.base_Operation.ownedParameter->isEmpty() and
        self.base_Operation.raisedException->isEmpty()
   ```

2. A formal action does not redefine another operation.

```
context UP4IS::FormalAction
inv: self.base_Operation.redefinedOperation->isEmpty()
```

3. A formal action does not define post, pre, and body conditions.

```
context UP4IS::FormalAction
inv: self.base_Operation.postCondition->isEmpty() and
     self.base_Operation.preCondition->isEmpty() and
     self.base_Operation.bodyCondition->isEmpty()
```

C.4 Interaction

1. An interaction is defined by binding an IT.

```
context UP4IS::Interaction
inv: let c : Collaboration = self.base_Collaboration
     in c.templateBinding->size() = 1 and
     let it : TemplateableElement = c.templateBinding
               ->first().signature.template
     in if it.oclIsTypeOf(Collaboration) then
          not it.oclAsType(Collaboration).
            extension_InteractionTemplate.
            oclIsUndefined()
        else
          false
        endif
```

2. For each formal action of the referenced IT exists an actual substitute. This constraint guarantees left-totalness for an interactions “is-replaced-by” relation $\mathcal{A}$ (cf. Definition 4).

```
context UP4IS::Interaction
inv: let c : Collaboration = self.base_Collaboration,
         binding : TemplateBinding = c.templateBinding
                    ->first(),
         it : TemplateableElement = binding.signature.
               template
     in it.ownedTemplateSignature.parameter
          ->forAll(p | binding.parameterSubstitution
            ->exists(sub | p = sub.formal))
```

3. An actual action binds exactly one formal action. This constraints guarantees left-uniqueness for an interactions “is-replaced-by” relation $\mathcal{A}$ (cf. Definition 4).

```
context UP4IS::Interaction
inv: let c : Collaboration = self.base_Collaboration,
         binding : TemplateBinding = c.templateBinding
                    ->first(),
         it : TemplateableElement = binding.signature.
               template
```

```
        in binding->collect(parameterSubstitution)
             ->forAll(s1, s2 | s1.actual = s2.actual implies
                 s1.formal = s2.formal)
```

4. A formal action is bound by exactly one actual action. This constraints guarantees right-uniqueness for an interactions “is-replaced-by” relation $\mathcal{A}$ (cf. Definition 4).

```
context UP4IS::Interaction
inv: let c : Collaboration = self.base_Collaboration,
         binding : TemplateBinding = c.templateBinding
                     ->first(),
         it : TemplateableElement = binding.signature.
                template
     in binding->collect(parameterSubstitution)
          ->forAll(s1, s2 | s1.formal = s2.formal implies
              s1.actual = s2.actual)
```

5. An interaction is used exactly once for an interaction use as it is uniquely defined for a concrete service.

```
context UP4IS::Interaction
inv: let iuses : Set(CollaborationUse) =
           CollaborationUse.allInstances()
              ->select(i |
                  not i.extension_InteractionUse.
                  oclIsUndefined())
     in iuses->forAll(i1, i2 | i1.type = i2.type implies
          i1 = i2)
```

6. Actual actions of an interaction are compatible to formal actions of the underlying IT. This constraint is an inherited standard constraint of a UML template parameter substitution [OMG09b, p. 630] and hence not replicated here. The constraint enforces an IT's substitution predicate c (cf. Definition 3).

C.5 InteractionTemplate

1. An interaction template has at least one template parameter.

```
context UP4IS::InteractionTemplate
inv: let c : Collaboration = self.base_Collaboration
     in c.isTemplate() and
        c.ownedTemplateSignature.parameter->notEmpty()
```

2. All template parameters represent formal actions.

```
context UP4IS::InteractionTemplate
inv: let s : TemplateSignature = self.base_Collaboration.
              ownedTemplateSignature
     in s.parameter->forAll(p |
            if p.oclIsTypeOf(Operation) then
              not p.oclAsType(Operation).
                extension_FormalAction.oclIsUndefined()
```

```
                    else
                      false
                    endif )
```

3. An interaction template contains exactly two interaction roles having different types.

```
context UP4IS::InteractionTemplate
inv: let roles : Set(TypedElement) =
              self.base_Collaboration.collaborationRole
     in roles->size() = 2 and
        roles->forAll(r1, r2 | r1.type = r2.type implies
          r1 = r2)
```

4. At least one of the interaction roles is explicitly typed. If an interaction role is typed, its type is an interface. Note, that the first part of this constraint is enforced by the previous constraint as not both role types can be undefined at the same time.

```
context UP4IS::InteractionTemplate
inv: let roles : Collection(TypedElement) =
              self.base_Collaboration.collaborationRole
     in roles->forAll(r | not r.type.oclIsUndefined()
          implies r.type.oclIsTypeOf(Interface))
```

5. An interaction role's interface references only formal actions of the nesting interaction template as its operations.

```
context UP4IS::InteractionTemplate
inv: let it : Collaboration = self.base_Collaboration,
         formal : Set(Operation) = it.
                    ownedTemplateSignature.
                    parameter->asSet(),
         types : Set(Interface) = it.collaborationRole
                   ->collect(type)->asSet()
     in types->forAll(t | t.ownedOperation
          ->forAll(o | formal->includes(o)))
```

6. A formal action is associated to exactly one interaction role via a role's interface. This constraint ensures the binding of each formal action to an interaction role as its destination, as defined by an IT's “is-destination-of” relation D (cf. Definition 3).

```
context UP4IS::InteractionTemplate
inv: let it : Collaboration = self.base_Collaboration,
         formal : Set(Operation) = it.
                    ownedTemplateSignature.
                    parameter->asSet(),
         types : Set(Interface) = it.collaborationRole
                   ->collect(type)->asSet()
     in formal->forAll(f | types->select(t |
          t.ownedOperation->includes(f))->size() = 1)
```

7. All connectors within an interaction template connect all interaction roles and no other elements. There exists at least one such connector.

```
context UP4IS::InteractionTemplate
inv: let it : Collaboration = self.base_Collaboration,
     in it.ownedConnector->notEmpty() and
          it.ownedConnector->forAll(c |
            let roles : Collection(ConnectableElement) =
                  c.end->collect(role)
            in roles->includesAll(it.collaborationRole) and
               it.collaborationRole->includesAll(roles) )
```

8. A formal action is reflected by a dedicated connector having the same name. There exist no other connectors.

```
context UP4IS::InteractionTemplate
inv: let it : Collaboration = self.base_Collaboration,
         formal : Set(String) = it.ownedTemplateSignature.
                    parameter.name,
     in it.ownedConnector->collect(name)
          ->forAll(c | formal->includes(c))
```

9. An interaction template has a behaviour specification in terms of an associated UML sequence diagram whose lifelines correspond to the template's collaboration roles and the template's formal actions are reflected by synchronous and asynchronous messages, transmitted via the collaboration's connectors.

```
context UP4IS::InteractionTemplate
inv: let it : Collaboration = self.base_Collaboration,
         roles : Set(ConnectableElement) =
                   it.collaborationRole->asSet(),
         connectors : Set(Connector) =
                       it.ownedConnector->asSet()
     in it.ownedBehavior->exists(b |
          if b.oclIsTypeOf(Interaction) then
            let i : Interaction = b.oclAsType(Interaction)
            in (
              let lifelines : Set(String) =
                    i.lifeline->collect(represents)->asSet()
              in lifelines->includesAll(roles) and
                 roles->includesAll(lifelines)
            ) and (
              i.message->forAll(m | connectors
                 ->includes(m.connector))
            )
          else
            false
          endif )
```

C.6 InteractionUse

1. An interaction use must reference a defined interaction.

```
context UP4IS::InteractionUse
inv: let i : Collaboration =
            self.base_CollaborationUse.type
          in not i.extension_Interaction.oclIsUndefined()
```

2. An interaction use specifies at least one target technology.

```
context UP4IS::InteractionUse
inv: self.realisation->notEmpty()
```

3. Every interaction role of the referenced interaction is bound to exactly one service role. This constraint is an inherited standard constraint of a UML collaboration use [OMG09b, p. 171] and hence not replicated here.

4. Within an interaction use, an interaction role of the referenced interaction is bound to a service role by a dedicated role binding. Additionally, both interaction roles are bound to different service roles.

```
context UP4IS::InteractionUse
inv: let b : Collection(Dependency) =
               self.base_CollaborationUse.roleBinding
     in b->forAll(client->size()=1 and supplier->size()=1)
        and b->forAll(b1, b2 |
          b1.client->first() = b2.client->first() implies
          b1.supplier->first() = b2.supplier->first())
```

C.7 OnewayAction

1. A oneway action can only be substituted by actual actions defining only input messages or no messages at all. For an actual action a the constraint is similar to testing for $\mathcal{T}_{out}(a) = \emptyset$.

```
context UP4IS::OnewayAction
inv: self.getActualActions()
       ->forAll(o | o.ownedParameter
         ->forAll(p | p.direction = #in))
```

C.8 Service

1. At least two service roles are defined for a service.

```
context UP4IS::Service
inv: let s : Collaboration = self.base_Collaboration
     in s.collaborationRole->size() >= 2
```

2. Only interaction uses are used as role connectors within a service.

```
context UP4IS::Service
inv: let c : Collaboration = self.base_Collaboration
     in c.ownedConnector->isEmpty() and
        c.collaborationUse->notEmpty() and
        c.collaborationUse->forAll(
          not extension_InteractionUse.oclIsUndefined() )
```

3. Interaction uses within a service connect only service roles of the same service and bind all interaction roles. This constraint is an inherited standard constraint of a UML collaboration use [OMG09b, p. 171] and not replicated here.

4. Every service role is connected to at least one interaction use within its enclosing service.

```
context UP4IS::Service
inv: let svc : Collaboration = self.base_Collaboration,
         roleBindings : Set(Dependency) =
           svc.collaborationRole->collect(roleBinding)
           ->flatten()->asSet()
     in svc.collaborationRole->forAll(r | roleBindings
           ->intersection(r.clientDepenedency)->notEmpty() )
```

C.9 ServiceComponent

1. A service component defines at least one service port.

```
context UP4IS::ServiceComponent
inv: self.base_Component.ownedPort
       ->exists(not extension_ServicePort.oclIsUndefined())
```

C.10 ServicePort

1. A service port is bound to exactly one service role via a service use.

```
context UP4IS::ServicePort
inv: bindings : Collection(Dependency) =
       CollaborationUse.allInstances()
       ->select(not extension_ServiceUse.oclIsUndefined())
       ->collect(roleBinding)->flatten()
     in bindings->select(client = self.base_Port)
          ->size() = 1
```

C.11 ServiceUse

1. A service use references a service.

```
context UP4IS::ServiceUse
inv: let i : Collaboration = self.base_CollaborationUse.type
     in not i.extension_Service.oclIsUndefined()
```

APPENDIX D

Target Mappings

D.1 Handling Synchronous and Asynchronous Actions

The exemplified target mappings in Section 7.1.2 of the presented IT library distinguish between synchronous and asynchronous actions. While CORBA supports both kinds of actions, i.e., mapping asynchronous actions to oneway operations (cf. Listing D.1), Java does solely feature synchronous invocation semantics. Asynchronous actions will be supported by mapping them to native method calls and providing a delegate object to detach them explicitly using the executor service of the Java concurrent package (cf. Listing D.2 and D.2). This delegate is part of the target adaptor and thus is transparent to the service component.

```
interface Role {
  void sync();
  oneway void asynch();
};
```

Listing D.1: Mapping synchronous *sync*() and asynchronous *async*() actions to CORBA.

```
public interface Role {
  public void sync();
  public void async();
}
```

Listing D.2: Mapping synchronous *sync*() and an asynchronous *async*() actions to Java.

```
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledThreadPoolExecutor;

public class RoleImpl implements Role {

  private Role fDelegate;

  private final static ScheduledExecutorService
    fExecutorService = new ScheduledThreadPoolExecutor(10);

  ...

  public void sync() {
    fDelegate.sync();
  }

  public void async() {
    fExecutorService.get().execute(new Runnable() {
        @Override
        public void run() {
          fDelegate.async();
        }
      });
  }
}
```

Listing D.3: The proxy object to detach an asynchronous *async*() actions in Java.

D.2 Message Type Mappings

For the presented case study, we specified a number of type mappings between UML and Java and CORBA. As throughout the thesis the described model transformations focus on the interaction semantics and their resulting component interfaces, the transformation of an action's message types to concrete programming languages or target platforms remain opaque. This gap is addressed in this section. However. the presented type mappings do not represent a mandatory standard but serve for demonstration purposes only. We distinguish between primitive types which can easily be converted between different technologies by pure value copying, and complex types which need further processing upon conversion.

D.2.1 Primitive Message Type Mappings

The examples within this thesis reference a set of primitive data types defined by UML (cf. [OMG09b, pp. 616-621]). These types are used within UML models for interactions and services. Table D.1 lists the mappings for these types to data types defined Java and CORBA IDL.

UML Primitive Type	Java/Java RMI Type	CORBA IDL Type
Integer	int	long
String	java.lang.String	string
Boolean	boolean	boolean

Table D.1: Mapping primitive message types.

D.2.2 Complex Message Type Mappings

Unfortunately, there is no "standard" way of modelling complex data types with UML, e.g., structures, unions, arrays, or recursive ones like lists and trees. Hence, there is also no default mapping for such data types to programming languages or middleware concepts. This section presents proposals of how to model some of such data types in UML and how they are converted to concepts of Java (including Java RMI) and CORBA as these technologies are used within the presented thesis.

Homogeneous Composite Data Types

Homogeneous composite data types represent collections of a fixed number of indexed elements of a common data type. For example, *arrays* are a typical representative of this class of data types.

UML. In UML, arrays can be expressed by defining a uniform lower and upper multiplicity for attribute/parameter types within classifiers or operations, respectively [OMG09b, pp. 49, 103, 120, and 136].

Java/Java RMI. Java directly supports the concept of arrays. For Java RMI, a proper serialisation strategy has to be provided for the underlying data type of the arrays elements.

For primitive types and data types which are derived from generic types of the Java standard library, there exist default serialisation algorithms.

CORBA. Like Java, CORBA defines a standard data type for arrays including respective marshalling strategies [OMG08a, p. 245].

Heterogeneous Composite Data Types

Heterogeneous composite data types are data types which are composed of a fixed set of named elements with possibly pairwise varying data types. One distinguishes between data types containing all defined sub-elements at the same time, referred to as *structures*, or data types which contain only one element out of the defined elements at a time, referred to as *unions*.

UML. In UML, structures can be modelled by classes [OMG09b, p. 49]. The structure's elements are then modelled as public class attributes. Such attributes must be the only owned elements of a class. Thus, the class can not define any additional elements like operations. For unions, a discriminant must be modelled explicitly to distinguish the current stored value type.

Java/Java RMI. In Java, a structure is typically mapped as two distinct parts: an interface which defines accessor methods (get/set) for each element of the structure, and a class which implements this interfaces. Thus, by generally programming against the interface and only referring to the class when one needs to get an instance for such an interface, the source code becomes robust against changes in the underlying implementation/representation of the mapped structure. Again, if non-standard types are used for elements, special care needs to be taken into account to ensure serialisability of the data type. Like in UML, a discriminant must be added for unions to explicitly distinguish the current stored value type.

CORBA. CORBA defines native *struct* and *union* data types and their respective marshalling [OMG08a, pp. 241–242].

Recursive Data Types

Recursive data types are data types which contain elements of their own type. Common examples are lists and trees. The main consequence when handling such data types is, that one has no a priori knowledge about the size or length of "values"/instances of such data types.

UML. In UML, recursive references are either modelled as class attributes having the type of the contained class, or by explicitly using an *association* relation to the class itself [OMG09b, p. 38].

Java/Java RMI. In Java, one can either use generic data types from the Java standard library, e.g., Lists, or define one's own type in the same manner as defining structures in Java, reusing the structures type for its elements.

CORBA. For CORBA, one must distinguish between lists and other recursive structures. For lists, one can use the predefined Sequence data type [OMG08a, p. 245]. For other recursive data types one defines a structure containing elements of the surrounding structures type.

Consequences of Complex Message Types

In UML, the modelling of complex data types is primarily given by a "well-defined" usage of classes in class diagrams. Hence, one should define an appropriate UML profile which offers special constraining stereotypes (cf. Section 4.6) to enforce such well formedness on the one hand, and to make the class' semantics as data type visually explicit.

Furthermore, in contrast to primitive data types, complex data types need special handling when being converted from one representation/technology to another. This is because of varying concepts of how complex types are expressed by primitive elements of the underlying technologies. Furthermore, in the context of communicating systems, the marshalling, i.e., the serialisation of instances (values) of data types, plays an important role. Thereby, different technologies apply different marshalling strategies that are commonly implemented as part of the data type representation itself. Hence, when crossing technology boundaries in distributed systems, data values have to be translated between different representations. Within our presented service modelling approach, it is the task of the target adaptors (cf. Section 5.1.5) to handle such type conversions if necessary.

APPENDIX E

Run-time Framework

E.1 The Open Services Gateway Initiative

The *Open Services Gateway Initiative* (*OSGi*)[1] alliance is an vendor independent consortium which defines and promotes a universal middleware for a "dynamic module system for Java", referred to as OSGi Service Platform. If not mentioned otherwise, we just use OSGi for short when referencing the OSGi Service Platform. OSGi defines a flexible execution environment for Java based software components which offer or consume functionality/resources to or from other components. The OSGi framework itself is only described by a set of Java interfaces which provide a public API to access its core services, currently in release 4 [OSG11]. The OSGi alliance does not provide an own implementation of these interfaces. Instead, there exist a number of OSGi compliant commercial and open source products, e.g., IBM's Service Management Framework[2], KnopflerFish[3], and the Eclipse Foundation's Equinox[4] which is used for our case studies. It is guaranteed, that OSGi compliant software components run on each of these platforms without modifications.

Figure E.1 depicts the overall system architecture defined through OSGi. As already indicated, the framework is based on a Java virtual machine as back-end. On the front end, it provides means to manage software components, referred to as *bundles*. OSGi defines a number of standard bundles which offer functionality which is typically required by a majority of software systems. For instance, the logger bundle is such a component. A bundle is physically represented in the form of a *Java Archive* (*JAR*) file which is loaded by OSGi. A bundle can offer functionality to or may require functionality from other bundles. The resulting dependencies between bundles are automatically resolved by the underlying framework. Furthermore, a bundle can be dynamically loaded, started, stopped, and unloaded during run-time. Of course, if one bundle depends on a bundle which is stopped, the dependent bundle will be stopped as well.

[1]`http://www.osgi.org`.
[2]`http://www-306.ibm.com/software/wireless/smf`
[3]`http://www.knopflerfish.org`
[4]`http://www.eclipse.org/equinox`

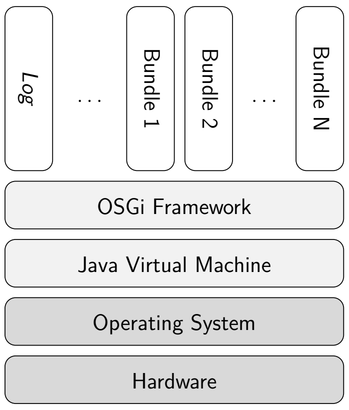


Figure E.1: The OSGi architecture.

E.2 Service Components and Target Adaptors

We base our prototyping system for the presented case studies on the dynamic run-time features and packaging concepts of OSGi. Therefore, all deployable system artifacts which are manual or automatic outcomes of the presented service development process are encapsulated in OSGi bundles. We distinguish between three kinds of bundles:

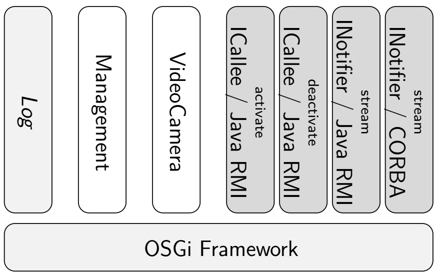
- Service Components,
- Target Adaptors, and
- Service Management.

Service components are implementations of service roles as specified by component models as part of service PIMs. A service component is registered at the service management bundle with its provided service roles and, for each of these roles, its bound interaction roles.

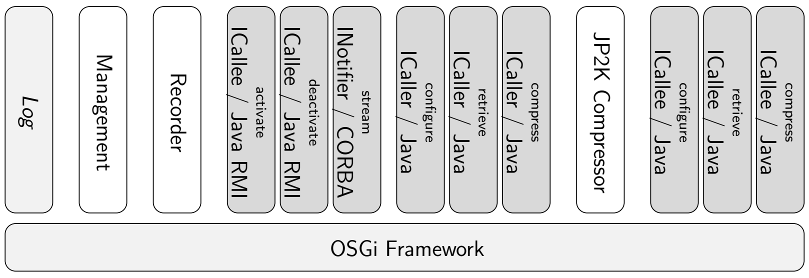
Target adaptors represent realisations of service interaction on concrete target platforms, e.g., Java RMI or CORBA. All necessary middleware elements and the adaptor classes themselves are wrapped in a target adaptor bundle, one for each interaction role - target platform pair. The bundle registers the implemented interaction role and the references target technology at the service management.

The service management bundle provides the run-time elements necessary to register and resolve service/interaction role implementations. It represents the infrastructure for our *Service Oriented Architecture* (*SOA*) environment. Upon registration of service and target adaptor bundles, it automatically connects service roles to their required interaction role implementations and registers the resulting endpoints with the service registry. Upon service role resolution, which is requested by a service component, it queries the registry for valid endpoints of the complementary interaction roles and establishes communication links if both interaction roles are realised on compatible platforms.

Figure E.2 depicts a sample system configuration for the video recording system. The system is split into two independent sub-systems. The one in (a) hosts only the camera service component and its three target adaptors. Thereby, the all interactions are realised with Java RMI technology. Hence, they can be accessed from a remote platform. Note, that in the given example, the video image stream interaction is also mapped to CORBA,



(a) The camera system.

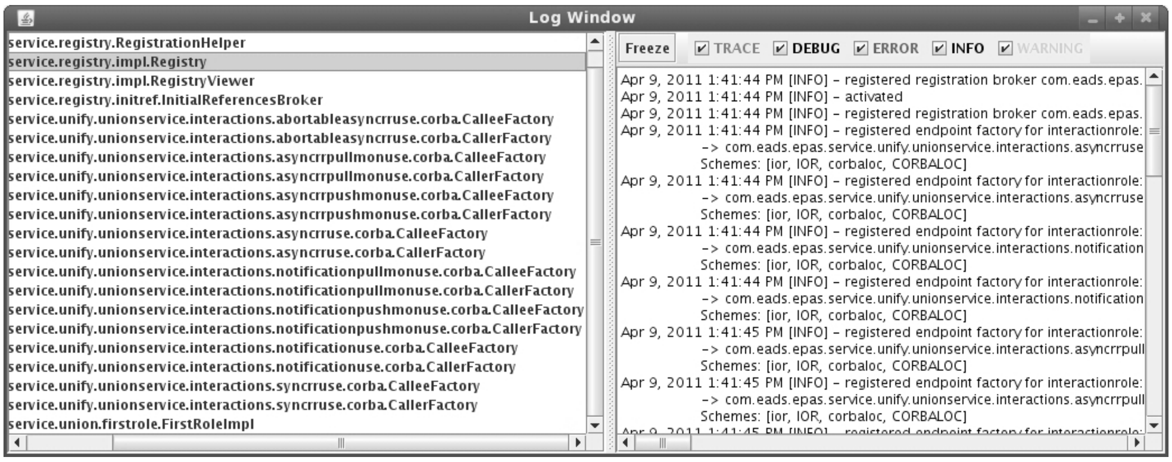


(b) The combined recorder compressor system.

Figure E.2: An exemplified deployment configuration for the video recording system based on OSGi (OSGi components light gray, service management/components white, and target adaptors dark gray).

which makes it accessible via two different communication paths using varying platforms. The second sub-system hosts the rest of the video recording system, i.e., the recorder and the JP2K compressor service components. The recorder interacts with the camera on the remote system via Java RMI for camera control and via CORBA for image reception. As the JP2K compressor component is local with respect to the recorder component, both components communicate directly via native Java adaptors avoiding indirection due to additional middleware components.

Figure E.3 depicts the prototyped run-time monitoring tool which supports for investigation of registered services, components and target adaptors.



```
service.registry.RegistrationHelper
service.registry.impl.Registry
service.registry.impl.RegistryViewer
service.registry.initref.InitialReferencesBroker
service.unify.unionservice.interactions.abortableasyncrruse.corba.CalleeFactory
service.unify.unionservice.interactions.abortableasyncrruse.corba.CallerFactory
service.unify.unionservice.interactions.asyncrrpullmonuse.corba.CalleeFactory
service.unify.unionservice.interactions.asyncrrpullmonuse.corba.CallerFactory
service.unify.unionservice.interactions.asyncrrpushmonuse.corba.CalleeFactory
service.unify.unionservice.interactions.asyncrrpushmonuse.corba.CallerFactory
service.unify.unionservice.interactions.asyncrruse.corba.CalleeFactory
service.unify.unionservice.interactions.asyncrruse.corba.CallerFactory
service.unify.unionservice.interactions.notificationpullmonuse.corba.CalleeFactory
service.unify.unionservice.interactions.notificationpullmonuse.corba.CallerFactory
service.unify.unionservice.interactions.notificationpushmonuse.corba.CalleeFactory
service.unify.unionservice.interactions.notificationpushmonuse.corba.CallerFactory
service.unify.unionservice.interactions.notificationuse.corba.CalleeFactory
service.unify.unionservice.interactions.notificationuse.corba.CallerFactory
service.unify.unionservice.interactions.syncrruse.corba.CalleeFactory
service.unify.unionservice.interactions.syncrruse.corba.CallerFactory
service.union.firstrole.FirstRoleImpl
```

```
Apr 9, 2011 1:41:44 PM [INFO] – registered registration broker com.eads.epas.
Apr 9, 2011 1:41:44 PM [INFO] – activated
Apr 9, 2011 1:41:44 PM [INFO] – registered registration broker com.eads.epas.
Apr 9, 2011 1:41:44 PM [INFO] – registered endpoint factory for interactionrole:
              -> com.eads.epas.service.unify.unionservice.interactions.asyncrruse
              Schemes: [ior, IOR, corbaloc, CORBALOC]
Apr 9, 2011 1:41:44 PM [INFO] – registered endpoint factory for interactionrole:
              -> com.eads.epas.service.unify.unionservice.interactions.asyncrruse
              Schemes: [ior, IOR, corbaloc, CORBALOC]
Apr 9, 2011 1:41:44 PM [INFO] – registered endpoint factory for interactionrole:
              -> com.eads.epas.service.unify.unionservice.interactions.notification
              Schemes: [ior, IOR, corbaloc, CORBALOC]
Apr 9, 2011 1:41:44 PM [INFO] – registered endpoint factory for interactionrole:
              -> com.eads.epas.service.unify.unionservice.interactions.notification
              Schemes: [ior, IOR, corbaloc, CORBALOC]
Apr 9, 2011 1:41:45 PM [INFO] – registered endpoint factory for interactionrole:
              -> com.eads.epas.service.unify.unionservice.interactions.asyncrrpull
              Schemes: [ior, IOR, corbaloc, CORBALOC]
Apr 9, 2011 1:41:45 PM [INFO] – registered endpoint factory for interactionrole:
              -> com.eads.epas.service.unify.unionservice.interactions.asyncrrpull
              Schemes: [ior, IOR, corbaloc, CORBALOC]
Apr 9, 2011 1:41:45 PM [INFO] – registered endpoint factory for interactionrole:
```

Figure E.3: The prototyped run-time monitoring tool.

APPENDIX F

Service/Interaction Role Interface Mappings

F.1 Video Capturing Service

F.1.1 Java Mappings

```
package videocapturing.activate;

public interface ICaller {
  // empty
}

public interface ICallee {
  public boolean start();
}
```

Listing F.1: Interaction role Java interfaces for activate.

```
package videocapturing.deactivate;

public interface ICaller {
  // empty
}

public interface ICallee {
  public boolean stop();
}
```

Listing F.2: Interaction role Java interfaces for deactivate.

```
package videocapturing.stream;

public interface INotifier {
  // empty
}

public interface INotifyee {
  public void nextFrame(final int time, final IImage img);
}
```

Listing F.3: Interaction Java role interfaces for stream.

```
package videocapturing;

public interface IClient extends
   videocapturing.activate.ICaller,
   videocapturing.deactivate.ICaller,
   videocapturing.stream.INotifyee {
  // empty
}

public interface ICamera {
   videocapturing.activate.ICallee,
   videocapturing.deactivate.ICallee,
   videocapturing.stream.INotifier {
  // empty
}
```

Listing F.4: Service role Java interfaces for video capturing.

F.1.2 CORBA Mappings

```
module videocapturing {
module activate {

    interface ICaller {
        // empty
    }

    interface ICallee {
        bool start();
    }

}; // activate
}; // videocapturing
```

Listing F.5: Interaction role CORBA interfaces for activate.

```
module videocapturing {
module deactivate {
    interface ICaller {
        // empty
    }

    interface ICallee {
        bool stop();
    }

}; // deactivate
}; // videocapturing
```

Listing F.6: Interaction role CORBA interfaces for deactivate.

```
module videocapturing {
module stream {

    interface INotifier {
        // empty
    }

    interface INotifyee {
        oneway void nextFrame(in long time, in IImage img);
    }

}; // stream
}; // videocapturing
```

Listing F.7: Interaction CORBA role interfaces for stream.

F.2 Image Compression Service

F.2.1 Java Mappings

```
package imagecompression.configure;

public interface ICaller {
  // empty
}

public interface ICallee {
  public boolean setMode(final CMode mode);
}
```

Listing F.8: Interaction role Java interfaces for configure.

```
package imagecompression.retrieve;

public interface ICaller {
  // empty
}
```

```
public interface ICallee {
  public CMode getMode();
}
```

Listing F.9: Interaction role Java interfaces for retrieve.

```
package imagecompression.compress;

public interface ICaller {
  public void compressed(final CImage img);
  public void progress(final int progress);
}

public interface ICallee {
  public void compress(final IImage img);
}
```

Listing F.10: Interaction Java role interfaces for compress.

```
package imagecompression;

public interface IClient extends
   imagecompression.configure.ICaller,
   imagecompression.retrieve.ICaller,
   imagecompression.compress.ICaller {
  // empty
}

public interface ICamera {
   imagecompression.configure.ICallee,
   imagecompression.retrieve.ICallee,
   imagecompression.compress.ICallee {
  // empty
}
```

Listing F.11: Service role Java interfaces for image compresssion.

F.2.2 CORBA Mappings

```
module imagecompression {
module configure {

    interface ICaller {
        // empty
    }

    interface ICallee {
        bool setMode(in CMode mode);
    }

}; // configure
}; // imagecompression
```

Listing F.12: Interaction role CORBA interfaces for configure.

```
module imagecompression {
module retrieve {

    interface ICaller {
        // empty
    }

    interface ICallee {
        CMode getMode();
    }

}; // retrieve
}; // imagecompression
```

Listing F.13: Interaction role CORBA interfaces for retrieve.

```
module imagecompression {
module compress {

    interface ICaller {
        oneway void compressed(in CImage img);
        oneway void progress(in long progress);
    }

    interface ICallee {
        oneway void compress(in CImage img);
    }

}; // compress
}; // imagecompression
```

Listing F.14: Interaction CORBA role interfaces for compress.

APPENDIX G

Theses

1. Service-orientated design and model-driven development principles can help to reduce design complexities within the development of mission critical embedded systems such as avionics.

2. Services are primary characterised by the interactions occurring between service participants upon service establishment.

3. Current approaches for combining service-orientation and model-driven development in system design processes typically restrict service interaction to fixed sets of primitives which are tightly coupled to specific target platforms.

4. Restricting the set of service interaction patterns within the development process reduces platform independence of service specifications.

5. Service interactions can be described independent from the target platforms they are to be realised on.

6. Concrete interactions within services can be generalised to sets of formal Interaction Templates which capture the specific characteristics of the underlying interaction semantics of service interactions in a generic manner.

7. Within an model-driven service development process, Interaction Templates serve as building blocks for service specifications via template instantiation and composition.

8. Required Interaction Templates may not be defined in advance to service modelling but can be specified on demand by the same process used to specify services themselves.

9. Using Interaction Templates for service modelling increases platform independence of service specifications.

10. Grounding concrete service interactions which are derived from Interaction Templates on target platforms is a fully automatable process.

11. For the automatic grounding of concrete service interactions on target platforms it is sufficient to provide generic mapping rules operating on the interactions' underlying Interaction Templates, unaffected by the concrete context of the templates' instantiations in services.
12. Modelling services based on Interaction Templates is an MDA conforming process which can be implemented through standard UML concepts and tools.