Model-Driven Semantic Integration of Service-Oriented Applications

Stanislav Vassilev Pokraev

Enschede, The Netherlands, 2009

Novay PhD Research Series, No. 025 (Novay/PRS/025)
CTIT Ph.D.-Thesis Series, No. 09-151
SIKS Dissertation Series, No 2009-29

Cover Design: Morskieft Ontwerpers, Enter
Book Design: Lidwien van de Wijngaert and Henri ter Hofte
Printing: Universal Press, Veenendaal, The Netherlands
Cover photo: "Atomium". Photo by Marc Lankhorst, 2008

Graduation committee:
Chairman, secretary: prof.dr.ir. A.J. Mouthaan (University of Twente)
Promotor: prof.dr.ir. R.J. Wieringa (University of Twente)
Co-promotor: prof.dr. M. Reichert (University of Ulm)
Assistant Promotor: dr.ir. M.W.A. Steen (Novay)
Members: prof.dr. C. Atkinson (University of Mannheim)
prof.dr.ir. G.J. Houben (Delft University of Technology)
prof.dr. M. Aiello (Rijksuniversiteit Groningen)
prof.dr. J. van Hillegersberg (University of Twente)
dr.ir. M.J. van Sinderen (University of Twente)

Novay PhD Research Series, No. 025 (Novay/PRS/025)
ISSN (print) 1877-8739; No. 025
ISSN (online) 1877-8747
ISBN 978-90-75176-49-0
Novay, P.O. Box 589, 7500 AN Enschede, The Netherlands
E-mail: info@novay.nl; Internet: http://www.novay.nl
Telephone: +31 (0)53-4850485; Fax: +31 (0)53-4850400

CTIT Ph.D.-Thesis Series, No. 09-151
ISSN 1381-3617; No 09-151
Centre for Telematics and Information Technology, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

SIKS Dissertation Series, No. 2009-29
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

MODEL-DRIVEN SEMANTIC INTEGRATION OF SERVICE-ORIENTED APPLICATIONS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof.dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 22 oktober 2009 om 15.00 uur

door
Stanislav Vassilev Pokraev
geboren op 06 januari 1972
te Sliven, Bulgarije

Dit proefschrift is goedgekeurd door:
prof.dr.ir. R.J. Wieringa (promotor), prof.dr. M. Reichert (co-promotor) en
dr.ir. M.W.A. Steen (assistant promotor)

Acknowledgements

This thesis is the result of research carried out over a period of five years. During this period, many people supported my work and helped me to bring it to a successful conclusion, and here I would like to express my gratitude.

First of all, I would like to thank my family and especially my wife Vania, for their unconditional support and outstanding belief in my success. Without you, it would not have been possible for me to go this far.

I present my words of gratitude to my promotor Roel Wieringa, my co-promotor Manfred Reichert and my assistant promotor Maarten Steen. Your continuous support played an essential role in helping to evolve my ideas and improve the quality of this dissertation. My special thanks go to Maarten for being such a great mentor and a very good friend. Maarten, you helped me through many crises and convinced me to keep going. I deeply appreciate your valuable advice on both professional and personal matters.

Two people played especially important roles in my development as a researcher: Rogier Brussee and Dick Quartel. Rogier, thank you for being my supervisor in the first year of my research. You did a tough job of convincing me that I should start from the problem and not from the solution. Dick, you helped me enormously in the last four years. I learnt a lot from you and I am deeply grateful for your unconditional support in both writing scientific articles and implementing proof-of-concepts demonstrators.

I would like to extend my gratitude to all my present and past colleagues from Novay, especially the ones I have worked with: Johan, Mark, Peter, Martin, Diederik, Henk, Mettina, and Willem. Special thanks to Bob Koehoorn for providing me with the data for my second validation case. I am also especially thankful to Ferial, Patrick and Marc - my people managers - for believing in me and providing me with space to do my research. Many thanks to the TAO group – Arjan, Cristian, Erwin, Ingrid, Lilia, Luit, Margit, Mark, Niels, Robert and Ynze – for sharing both the joy

and the frustration of being a PhD researcher. My deep gratitude to Maria and Sorin Iacob for being such nice colleagues, neighbours, and friends. I really enjoyed your company and support in all these years. Last but not least, special thanks to Andrew Tokmakoff for reviewing some chapters of this thesis and helping me to improve the written English.

I would like to thank the members of my defence committee: Professor Colin Atkinson, Professor Geert-Jan Houben, Professor Marco Aiello, Professor Jos van Hillegersberg, and Dr. Marten van Sinderen for devoting time to read my thesis and to participate in my defence. It is an honour to have you in this committee.

Finally, I would like to thank all my friends for supporting me outside the office and making my life more pleasant: Nikolay Kavaldjiev, Nikolay Diakov, Babbata, Emil Devedjiev, Ivan Kurtev, Chris, Zlatko, Andrew, Elfi, Seema, Rene, Pusho, Judi, Ivaylo, Dano, Lora, Julia, Boriana, George, Lilith, Sami, Lucy, Tony, Ina, Goran, Nikolay Dokovsky, Maya, Ivayla, Ulrich, Andreas, Giancarlo, Renata, Joao-Paulo, Patricia, Gabriele, Natasa and Teduh. I hope I have not forgotten anyone. Special thanks to my best friend Nikolay Kavaldjiev, for all the fun we had during these years and for making me so enthusiastic about digital photography.

Stanislav Pokraev
Enschede, July 2009

Contents

Part I.
Introduction

Chapter 1

Introduction

In this thesis, we propose a *method for the semantic integration of service oriented applications*. The distinctive feature of our method is that *semantically enriched service models* are employed at *different levels of abstraction* to deliver *flexible*, end-to-end integration solutions from business requirements to software implementation.

This chapter is organised as follows: Section 1.1 provides background and motivation for the research presented in this thesis. Section 1.2 defines the objective of the research and requirements for the proposed solution. Section 1.3 presents the research questions that guide this research. Section 1.4 present the adopted research methodology and the concrete research methods used to achieve the objective of the research. Section 1.5 summarises the contributions of this research. Finally, Section 1.6 presents the structure of the remainder of the thesis.

1.1 Background and Motivation

In the last decades, enterprises have been using an increasing number of different software applications to support their business processes. Nowadays, it is common, that a single enterprise uses hundreds of applications, developed by *different vendors*, running on *different operating systems* and using *different databases*. Examples of such applications are *Customer Relationship Management* (*CRM*), *Financial Accounting* (*FA*), *Enterprise Resource Planning* (*ERP*), *Digital Asset Management* (*DAM*) and *Logistics Information* (*LI*) systems. Besides, very often, an enterprise develops *custom applications* to support specific aspects of its product development or service provisioning. In addition, especially after mergers or acquisitions, *multiple applications with the same or overlapping functionality* are used in one enterprise.

1.1.1 Enterprise Application Integration

The need for *Enterprise Application Integration* (EAI) arose as enterprises started to recognise that integration of systems could, among other things, save time and money when developing a new product or service, increase the flexibility and adaptability of their overall business processes and improve the organization's decision-making capabilities.

In general, an *integrated system* is a collection of subsystems that *interact* to form a *whole*. It has *properties* that *emerge* due to the interaction of its subsystems. Enterprises integrate their applications because the emerging properties of the integrated system have *value* for them. Examples of such emerging properties are more efficient usage of the available enterprise resources, flexibility and adaptability of business processes, and increased market reach.

In general, we can distinguish two aspects of EAI – the *information* and *process* aspects:

- *Information aspect*. In many cases, an enterprise uses different systems or different instances of the same system to manage information about the same entity or phenomenon in the real world, for example, information about a particular customer or product. In this case, an EAI solution should take care that the information about this entity or phenomenon in all systems and instances is *mutually consistent*.

- *Process aspect*. Enterprises define their business as a sequence of activities that concern a specific business case, for example, handling of a purchase order. An EAI solution should ensure that all information systems, supporting the business process, are updated in the correct order as the business process progresses. This means, that the EAI solution should implement the *correct control and data flow* across the systems being integrated. Note that there is a duality between information and process aspects. That is, changes in the first often require changes in the second and vice versa. Therefore, an EAI solution should be able to capture such a relation and deal with it.

EAI is an extremely complex process. The reason is that the applications that have to be integrated *have not been designed to work together*, that is, they are *heterogeneous*, *autonomous* and *distributed* (*HAD*). *Heterogeneous* systems use different information models to capture the semantics of the business domain. *Autonomous* systems exchange data following their own interaction protocols independently from the interaction protocols of any other system. *Distributed* systems do not share common state and use different means to

update or retrieve this state. EAI is about *solving large-scale inter-disciplinary problems enabling multiple HAD systems to interoperate*.

The integration problem has three main aspects. The first aspect concerns *information* integration of the systems which is necessary due to the *heterogeneous* nature of the systems. The second aspect concerns *process* integration of the systems which is necessary due to the *autonomous* and *distributed* nature of the systems. Finally, the third aspect concerns the complexity of the *design*, *development* and *maintenance* of the integrated solution.

To address each problem aspect, three main approaches are proposed and presented in the following sections. Since the *problem aspects always occur together*, *the solutions* for them *must be combined*. In this thesis, we investigate how and to what extent these solutions can be combined.

As this is an introductory chapter, we will only briefly sketch the problem and the proposed solutions. For a more detailed presentation and discussion, the reader is referred to Chapter 3.

In the reminder of Section 1.1, we briefly sketch currently available solution directions that we consider to be relevant for solving (or mitigating) the presented EAI problem. In this thesis, we argue for the need of combining these solutions approaches and demonstrate how this can be achieved.

1.1.2 Service Orientation

Service orientation is currently considered to be a promising architectural approach for building EAI solutions. The service-orientation paradigm is characterised by the *explicit identification* and description of the *externally observable properties of a system* (e.g., a software application or a business process). Different systems can then be *linked*, based on the description of their external properties, *without any knowledge of their internal implementation details*.

To support service orientation, a great deal of effort is currently being invested in the standardization of service description languages known as *Web services* (WS[1]) standards.

Currently, WS languages only standardise the *syntax* of service descriptions. They do not provide means for defining the *semantics* of a service. This means that although syntactically correct, a service description still can define *unintended*, *admissible state of affairs* of the real world (cf. Figure 1-1).

[1] http://www.w3.org/2002/ws/

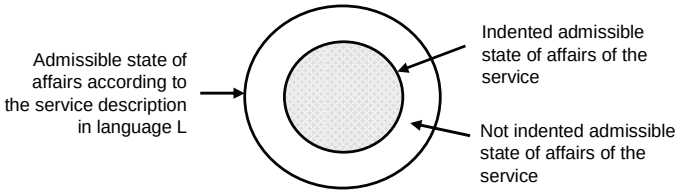


Figure 1-1
Service models and descriptions

Semantic service descriptions are even more important when integrating different systems. Two systems can only interoperate if they exchange data with *compatible meaning.* In addition, they can only achieve a desired effect if they exchange these data following *compatible interaction protocols*. By analyzing the service descriptions, a system integrator could conclude that the systems are interoperable. However, in reality this might not be the case (cf. Figure 1-2). The problem is known as the *false agreement* problem and is discussed in (Guarino, 1998).

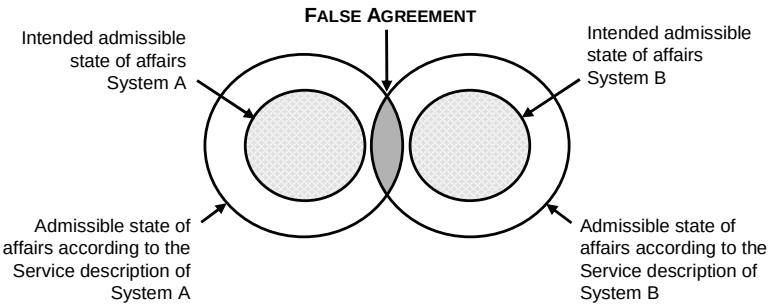


Figure 1-2
The false agreement problem

If a false agreement is not discovered early in an integration project (e.g., the design phase) it usually leads to the implementation of an incorrect integration solution. This, in turn, unnecessarily increases the cost of the integration project.

The lack of formal semantics in service descriptions makes it very difficult (sometimes even impossible) to use automatic reasoners to discover false agreements in the early phase of an integration project. In the worst case, even after extensive testing, an incorrect integration solution is completely implemented and deployed. This usually leads to very high costs to then repair the solution, or in some cases, even to loss of business. Note that in some cases, the cost of formalizing service descriptions can be higher than the benefit of automatically checking the correctness of the integration solution. However, domain standards can be formalised and used to build service descriptions, which can help to justify the cost of their formalization.

1.1.3 Knowledge Representation

Knowledge Representation (KR) enables the formal specification of service semantics. KR relies on *ontologies* - formal representation of consensual knowledge about some domain of interest.

Ontologies can provide different degrees of formalization as shown in Figure 1-3.

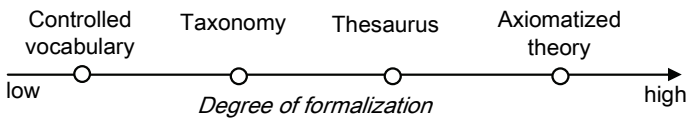


Figure 1-3 Degree of formalization of ontologies

Controlled vocabularies are the weakest form of ontology. A controlled vocabulary is an exhaustive list of terms with unambiguous and accurate definitions. The main objective of a controlled vocabulary is to prevent the use of ambiguous, meaningless or misspelled terms in service descriptions by defining them explicitly.

Taxonomies are subject-based classifications of the terms in some controlled vocabularies. Taxonomies classify the terms into hierarchy defining explicit "is subclass of" relationships between a term and other terms.

Thesauri are networked collections of controlled vocabularies with richer semantic relationships between terms. A thesaurus is an extension of a taxonomy allowing stating not only "is subclass of" relationships among terms but also, for instance, *equivalence*, *similarity* and *homographic* relationships.

Axiomatized theories are the strongest form of ontology. Similar to thesauri, an axiomatized theory allows specification of semantic relations among terms in controlled vocabularies as well as formal rules on how to construct complex terms and relationships. These formal rules enable inferencing of new knowledge and formal reasoning.

In order to formally define the semantics of a service, we need a formal ontology. This, in turn, enables the early discovery of false agreements and the automatic verification of integration solutions.

1.1.4 Model-Driven Architecture

Building an integration solution is a process of building a system that satisfies some integration requirements. Such a system is built by linking existing systems and, if necessary, compensating the mismatches between them by adding additional integration logic. Often, such integration logic is hard-coded in the solution implementation. That is, the logic to compensate the mismatches between the systems is deeply hidden in the

application code. In turn, this substantially increases the cost to maintain such integration solutions. For example, if the integration requirements change, it will take time and resources to discover the corresponding code and update it. The updated code must be re-tested which adds even more additional cost and delays. Furthermore, domain experts are usually only involved at the very early stages of an integration project, namely in the requirements elicitation phase. This makes the gap between requirements and the implementation wide and additionally complicates the integration process. To simplify the process, a good integration method should allow system integrators (both domain experts and software engineers) to *address* only a *limited set of concerns in a series of design steps*. This principle is known as *separation of concerns*.

Model Driven Architecture (*MDA*) has been proposed as a new paradigm for software development. In MDA, the separation of concerns is achieved by *specifying models at different level of abstraction*. Each of these models focuses on the characteristics of an entity (or phenomenon) that are considered essential for a certain purpose, while ignoring or discarding details that are considered irrelevant for the same purpose.

MDA consists of three basic principles. First, in MDA *the focus* of software development is *shifted* away from the *technology domain* to the *problem domain*. In this way, the solution is described using a language that is closer to the language used to describe the problem. This reduces the semantic gap between the problem and solution and decouples the solution from the problem enabling the reuse of the same solution for different problems. Second, MDA enables *automation* by mapping domain concepts to implementation technology by the means of (executable) *model transformations*. In this way, the models produced in the design phase of the project are not only used for documentation purpose but also for *code generation* and *requirements traceability*. Finally, MDA is based on *open standards*, which encourages the adoption of these standards by different vendors and thus reduces the heterogeneity.

1.2 Research Objective

The objective of this thesis is to investigate *to what extent and how SOA, KR and MDA can be combined to improve existing EAI approaches*. To the extent in which they can be combined, we provide *a method for the semantic integration of service-oriented applications*. More precisely, to address the shortcomings of existing EAI approaches, we define a number of requirements for our method. These requirements follow from the capabilities of the solution approaches presented in Section 1.1, which, in turn, are motivated by the stakeholders in the problem domain.

- *Requirement R1*. The method should provide for defining the integration solutions in terms of the problem domain, rather than in terms of solution technologies. This will enable the more active participation of domain experts, e.g., they could be involved in specifying and verifying the semantic relations among corresponding domain concepts and the correct order of system interactions. In turn, this will simplify the integration process and result in more correct solutions. The first reason is that active participation of domain experts can relieve software engineers from making decisions in domains beyond their competence. The second reason is that the models produced by the domain experts will reflect 'the reality' better since they are experts in the problem domain.

- *Requirement R2*. The integration method should enable the semantic integration of services. The current service description standards enable only the syntactic integration of systems. These standards do not provide means for semantic integration. Service descriptions specified using existing service description standards are ambiguous and do not capture the hidden assumptions made about systems. To enable different systems to interoperate correctly, an integration solution should not only compensate for mismatches in the data format of exchanged messages between the integrated systems, but also enforce the uniform interpretation and use of these messages.

- *Requirement R3*. The integration method should enable the *formal verification* of the integration solution. Currently, the correctness of an integration solution is verified at a very late stage by performing tests after the integration solution is implemented. This is an expensive and time-consuming process. A formal verification of a solution design will reduce the costs and decrease the time required to verify the solution. In addition, a formal verification could discover problems, which cannot be discovered in the testing phase. Note, that in some cases formal verification can be very difficult or not required. For that reason, this requirement for the method is optional.

- *Requirement R4*. The integration method should allow for *changes in the implementation technology*. This means that if the implementation technology changes, it should be possible to reuse the same abstract solution specification defined by the domain experts. This will reduce the cost and decrease the time to implement a change. The requirement R4 is universal, that is, it applies to all integration methods in general. The reason is that all EAI solutions require change of implementation technology at some point of time.

– *Requirement R5:* The integration method should allow for *changes of the business requirements*. This means that if business requirements change, only the abstract solution specification needs to be updated to reflect the new business requirements. It should be possible to generate a solution implementation from the updated abstract solution specification. This is also a universal requirement for all integration methods. To address constantly changing market demands and to stay competitive, enterprises constantly integrate new systems or change them. This, in turn, imposes new requirements on the existing integration solutions.

1.3 Research Questions

To achieve the objective of this research, a number of research questions need to be answered. Answering these questions will provide us with knowledge about the problem domain and the capabilities of existing solution approaches. This, in turn, will serve as an input to the design and validation of our solution.

– *Research question Q1*: What does interoperability mean? What does it mean for different systems to interoperate? Are there different levels of interoperability? What interoperability problems exist?

– *Research question Q2*: What are the current system integration approaches? What are their drawbacks? What technologies have been proposed to address these drawbacks? How do these technologies interact when used together?

– *Research question Q3*: How to model the semantics of a service? What aspects of services should be modelled and how? At which abstraction levels? How can we use these concepts to reason about a service?

– *Research question Q4*: What is necessary for two or more systems to interoperate? How can we formally check if two or more systems are interoperable?

– *Research question Q5*: How can two or more non-interoperable systems be integrated and how can such integration be achieved in a systematic way? Does such integration solve the drawbacks of existing integration approaches?

These questions guide the research presented in this thesis. In Section 1.6, we present the structure of this thesis and a table that relates the research questions to the chapters in which we provide answers to the questions.

1.4 Research Methodology

In this research, we try to solve two types of problems: *knowledge problem*s and *design problems* (Wieringa, 2006).

Someone has a *knowledge problem* if there is a difference between his current and desired knowledge states: that is, he wants to *know* something. Someone has a *design problem* if there is a difference between the current and desired state of the world that he wants to reduce: that is, he wants to *build* something or *change* something in the real world.

Our research methodology consists of three phases - *problem analysis*, *solution design* and *solution validation*.

In the first phase, we solve a knowledge problem, for example, we want to understand what interoperability means, what are the interoperability problems and what solutions there are for these problems. For that purpose, we perform three literature studies. In the first study, we analyse literature from different areas including artificial intelligence, database research and process integration to discover possible interoperability problems. In the second study, we analyse the problems of existing EAI approaches. Finally, in the third study, we analyse the currently proposed technologies for system integration.

In the second phase, the results from the first phase are used to design a new solution. In this phase, we solve a design problem, that is, we define a conceptual framework for service modelling and propose a method for the semantic integration of service oriented applications. Our goal is to improve existing EAI approaches.

Finally, in the third phase, we validate our solution by investigating its suitability for the problems discovered in the first phase. This is a knowledge problem since we want to gain knowledge about the properties of our solution, and the relation between the solution and the problems. Validation is achieved by applying our solution to solve two characteristic integration problems. The knowledge we gain in the validation phase is fed back to the solution design phase in order to improve the solution.

The research methodology is presented in Figure 1-4.

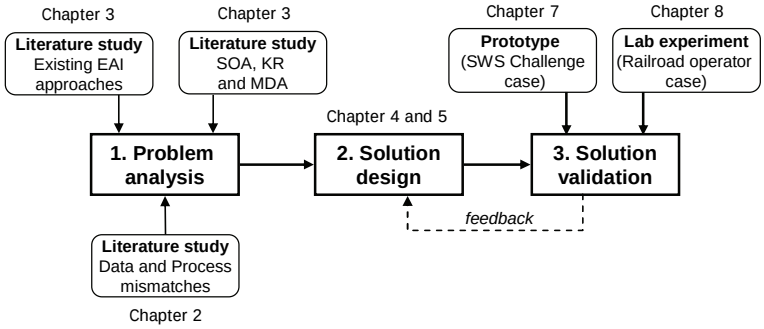


Figure 1-4 Research methodology

1.5 Contributions

The research, presented in this thesis, contributes to the effort in the area of *enterprise application integration*. Our main contributions are the following:

- We identify *common characteristics of interoperability* and give a definition of *interoperability*. Next, we identify *three different levels* of interoperability, namely, *syntactic*, *semantic* and *pragmatic* interoperability. At each of these levels, we identify *possible interoperability problems*.

- We identify *basic service properties* and define a *conceptual framework for service modelling* and *reasoning*. Our framework has a number of distinctive features. First, it is constructed from a *small number of basic concepts,* which are based on practice, but at the same time provide a powerful conceptual basis for service modelling. Second, the framework is *language-independent*, but at the same time, its basic concepts can be related to many of the popular languages used in the context of service design, analysis and implementation. Third, our framework is *domain-independent*, that is, no assumptions are made with respect to the type of services that should be modelled. Finally, the framework supports the *modelling of services at different abstraction levels*. More precisely, we identified three generic abstraction levels, namely, *service effect*, *choreography* and *orchestration*.

- We identify *necessary conditions* for interoperability and propose a *method for verification* whether a number of systems are interoperable. Our method enables the early discovery of false agreements and the automatic verification of integration solutions.

– We propose a *method for the semantic integration of service-oriented applications*. The key feature of our method is that *semantically rich* service models at *different abstraction levels* are employed to develop *flexible integration solutions* from business requirements to software implementation. The integration method allows for changes of the implementation technology as well as for changes of business requirements.

1.6 Outline of the Thesis

This thesis is organised as follows: Part II presents the problem analysis. It is organised in two chapters. In Chapter 2, we analyse the most cited interoperability definitions and derive *common characteristics of interoperability*. We use these common characteristics to define *what interoperability means* and identify three different levels of interoperability, namely, *syntactic*, *semantic* and *pragmatic* interoperability. Next, we study literature from different areas and identify *possible interoperability problems* at each of the interoperability levels.

In Chapter 3, we analyse existing EAI approaches and investigate their problems. Next, we study SOA, KR and MDA as these technologies have been proposed to solve the drawbacks of the current EAI approaches.

Part III presents our solution. It is organised into two chapters. In Chapter 4, we define a *conceptual framework for service modelling*. The purpose of the framework is to serve as a *common semantic meta-model* that enables the *description*, *integration* and *reasoning* about (integrated) *service-oriented applications*. Using the framework one can *model* the domain of a system, the interactions among its components and their relations, and *reason* whether these components are interoperable.

In Chapter 5, we present a *method for the semantic integration of service-oriented applications*. First, we identify *necessary conditions* for semantic and pragmatic interoperability of service-oriented applications. Next, we propose a *model-driven integration method* that uses *semantically enriched* service descriptions to deliver flexible integration solutions from business requirements to software implementation. Finally, we present a method to formally *verify* whether the proposed integration solution meets the identified conditions for interoperability.

Part IV presents the validation of our research. It is organised into four chapters. First, in Chapter 6, we provide *falsifiable claims* to prove whether our integration method satisfies the requirements defined in Section 1.2. Next, we provide *arguments for validity* of these claims by solving two characteristic integration cases presented in Chapter 7 and 8. Finally, in Chapter 9, we reflect upon both of the cases and present a *cross-case analysis*.

Finally, in Part V (Chapter 10) we summarise our *contributions* and list known *limitations*. We also provide directions for *future research*.

Figure 1-5 presents the relation of the research questions and the chapters of this thesis.

Figure 1-5
Relation of the research questions and the chapters of this thesis

	Part I	Part II		Part III		Part IV				Part V
	Chapter 1	Chapter 2	Chapter 3	Chapter 4	Chapter 5	Chapter 6	Chapter 7	Chapter 8	Chapter 9	Chapter 10
Research question Q1		x								
Research question Q2			x							
Research question Q3				x						
Research question Q4					x		x	x	x	
Research question Q5					x		x	x	x	

Part II.
State-of-the-art and Problem Analysis

Chapter 2

Interoperability and Interoperability Problems

The objective of this chapter is to give a *definition of interoperability* in the context of SOA, to present what *levels of interoperability* exist and to identify the *possible interoperability problems* at each of these levels. The chapter is organised as follows: In Section 2.1 we present the most cited definitions of interoperability and use them to derive some common characteristics of interoperability. In addition, we identify three levels of interoperability, namely *syntactic*, *semantic* and *pragmatic* interoperability. In Section 2.2, 2.3 and 2.4 we discus the problems that occur at each of these levels. Finally, in Section 2.5 we present our conclusions.

2.1 Interoperability

There have been many attempts to define what *interoperability* means. Below we present the most cited definitions and use them derive some common characteristics of interoperability.

- the ability to operate in conjunction (Oxford Dictionary, 2003)
- the ability of two or more systems or components to exchange information and to use the information that has been exchanged (IEEE, 1990)
- the capability to communicate, execute programs, or transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units (ISO, 2003)

- the condition achieved among communications-electronics systems or items of communications-electronics equipment when information or services can be exchanged directly and satisfactorily between them and/or their users (DoD, 2001)

- the ability to share and exchange information using common syntax and semantics to meet an application-specific functional relationship (ISO, 2000)

- the ability of two or more systems or components to exchange and use shared information (Open Group, 2000)

- the ability of systems to provide and receive services from other systems and to use the services so interchanged to enable them to operate effectively together (Open Group, 2000)

- the ability of information and communication technology (ICT) systems and of the business processes they support to exchange data and to enable the sharing of information and knowledge (EC-IDA, 2005).

According to the definitions given above, interoperability can be characterised by the following properties:

- involves *multiple (two or more) entities* (e.g., systems, components, units, forces, organizations)

- is ability to *interact* (e.g., to operate in conjunction, to communicate, to transfer data, to exchange information or knowledge, to provide and to accept services)

- requires *little or no knowledge* of the unique characteristics of the interacting entities

- is about *achieving* some *goal* (to operate effectively together, to meet an application-specific functional relationship, to exchange information or services satisfactorily)

In the context of SOA, systems interact using each other's services, that is, a system *provides services* to and *uses services* from other systems. Thus, in case of software systems, interoperability is *the ability of the software systems to use each other's software services*, i.e., to exchange data and use the exchanged data. In case of business systems, interoperability is the ability of the systems to use each other's business services, i.e., to provide business functions to each

other's and use the provided business functions. Services hide the unique characteristics of the systems that provide them, e.g., a software service hides the specific system implementation technology and a business service hides the internal company structure and the internal business processes. In addition, according to SOA paradigm, multiple systems can interact with little or no knowledge of each other's unique characteristics. Finally, using each other's services systems should be able to achieve some *goal*.

Based on the identified interoperability properties in the definitions given above we give the following definition of interoperability in the context of SOA:

Interoperability is the ability of multiple systems to use each other's services effectively.

When building an information system, the creator of the system decides what *entities* (or phenomena) from the real world should be *represented* in the system and which of their *properties* are important for the purpose of the system. Based on that, he or she defines a *language* to interact with the system. A language, according to (Morris, 1938), comprises three parts: *syntax*, *semantics* and *pragmatics*. *Syntax* is devoted to "the formal relations of signs to one another", *semantics* to "the relations of the signs to real world entities they represent", and *pragmatics* to "the relations of the signs to (human) interpreters". Using the distinction given by Morris, we define t*hree levels of interoperability*, respectively *syntactic*, *semantic* and *pragmatic interoperability*. In the following sub-sections, we elaborate upon each of these interoperability levels.

2.2 Syntactic Interoperability

The *syntax* of a language defines a *list of valid words* in the language (called *vocabulary*) and the *rules* that govern the way words combine into *sentences* (called *grammar*). A *parse tree* is a tree that represents the *structure* of a sentence according to some language grammar. The process of constructing a parse tree is called *parsing*.

Syntactic interoperability is concerned with ensuring that systems, involved in some communication use the same vocabulary and grammar to parse the exchanged sentences. *Syntactic interoperability problems* arise when the systems use incompatible vocabularies or grammars. This leads to inability to create a correct parse tree (or to construction of an incorrect parse tree) and inability to use the data in the exchanged sentences.

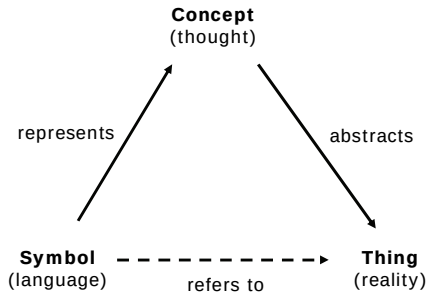
In Chapter 3, we present Web services as the most significant standardization efforts towards syntactic interoperability of services. Dealing

with syntactic interoperability problems is outside the scope of this thesis. For that reason, we only focus on semantic and pragmatic interoperability problems.

2.3 Semantic Interoperability

Semantics is concerned with the *meaning* of the syntactic constructs in a language. According to (Wood, 1985) semantics is "the meaning and the use of data". For our purpose, this definition is too general and not practical. Instead, in the context of information systems, semantics is defined as "a mapping from an object in an information system and a real-world object it represents". This definition is well-supported by the Ullmann's *meaning triangle* (Ullmann, 1972) which derives from (Ogden & Richards, 1923) and from (de Saussure, 1986) – the two most important theories that are the basis of the modern science of language.

Figure 2-6 The semantic triangle



The meaning triangle distinguishes between *things*, *concepts*, and *symbols*. A *thing* is any entity (or a relationship between two or more entities) in the real world. During our lives we learn to classify such real-world things into *abstract classes*, i.e., we derive *concepts* that *abstract* the real-world entities (or relationships between two or more entities) with similar characteristics. A concept is part of our "internal reality", i.e., it is a thought that only exists in our minds. In order to communicate, we need a *symbol* that *represents* the concept by the means of language and thus *refers* to the thing in the real world.

Concepts can be derived in two ways – by explicitly enumerating all the things that a concept abstracts, or by stating some properties that must be true for all things that a concept abstracts. In the first case, we say that a concept is defined by *extension*. In the second case, we say that a concept is defined by *intention*. A concept can be defined by extension if it abstracts a finite set of things. Infinite sets of things are always abstracted by intentional concept or by combining previously defined concepts.

Semantic interoperability is concerned with ensuring that a symbol has the same *meaning,* (i.e., refers to the same thing in the real world) for all systems that use this symbol in their languages. *Semantic interoperability problems* arise when different systems use *different symbols* to refer to the *same things* in the real world or use the *same symbol* to refer to *different things* in the real world. As explained earlier, a symbol refers to a thing in the real world *indirectly*, i.e., the symbol represents a concept that abstract the real-world thing. This means that semantic interoperability problems are either caused by *different abstraction* of the same real-world entities (or the relationships among them) or by *different representations* of the same concepts.

When integrating information systems, the system integrator cannot change the way in which the system creator has abstracted the real world entities. However, in some cases the system integrator can build a *mediator* that "*translates*" the symbols, exchanged between the systems. By *translation* we mean the process of interpreting a sentence sent by one system (according to the language of that system), and the production of a sentence (according to the language of the other system).

The semantic interoperability problems have been studied extensively in the context of databases, information systems and agent systems. Good classifications of the semantic problems can be found in (Sheth and Kashyap, 1992; Naiman and Ouksel, 1995; Goh, 1997; Visser, 1997; Klein, 2001). Using the knowledge built in the aforementioned areas, we present the possible semantic representation problems illustrated by simple examples.

Problem IP1. Different systems use the same symbol to represent concepts with disjoint meanings.

For example, one system uses the symbol "name" to represent the concept "a name of a place", whereas other system uses the same symbol to represent the concept "a name of a person" (cf. Figure 2-7).

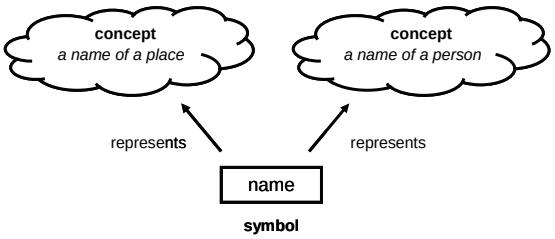
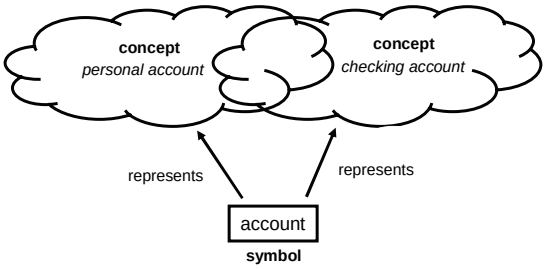


Figure 2-7 Same symbol, different concepts

Problem IP2. Different systems use the same symbol to represent concepts with overlapping meanings.

For example, one system uses the symbol "account" to represent the concept "personal account", whereas the other system uses the same symbol to represent the concept "checking account". Since not all personal accounts are checking accounts and not all checking accounts are personal accounts, in some cases the symbol "account" can refer to different entities in the real world (cf. Figure 2-8).

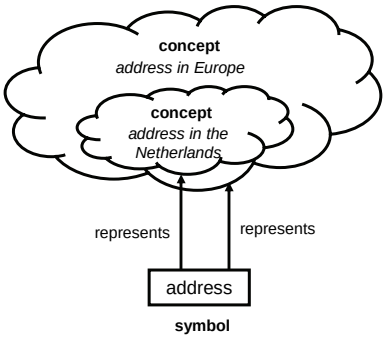
Figure 2-8
Same symbol, overlapping concepts



Problem IP3. Different systems use the same symbol to represent concepts with more general (or more specific) meanings

For example, one system uses the concept "address" to represent the concept "an address in the Netherlands", whereas other system uses the same symbol to represent the concept "an address in Europe" (including all addresses in the Netherlands)(cf. Figure 2-9).

Figure 2-9
Same symbol, more general (specific) concepts



Problem IP4. Different systems use different symbols to represent the same concept

For example, one system uses the symbol "customer" to represent the concept "someone that purchases goods or services" whereas other system uses the symbol "client" to represent the same concept (cf. Figure 2-10).

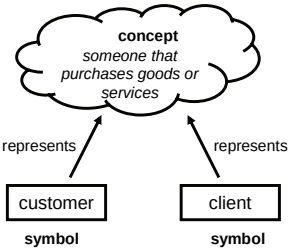


Figure 2-10
Different symbols, same concept

Problem IP5. Different systems use different symbols to represent concepts with overlapping meanings

For example, one system uses the symbol "employee" to represent the concept of "someone that works for a company" whereas other system uses the symbol "customer" to represent the concept of "someone that buys from a company" (cf. Figure 2-11). Since a customer can be an employee and an employee can be a customer, in some cases both symbols "employee" and "customers" can refer to the same entity in the real world.

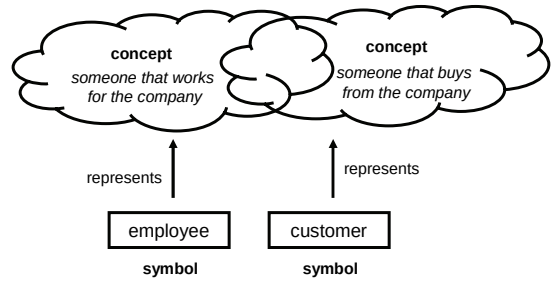


Figure 2-11
Different symbols, concepts with overlapping concepts

Problem IP6. Different systems use the different symbols to represent concepts with more general (or more specific) meanings

For example, one system may use the symbol "buyer" to represent the concept "buyer" whereas the other system may use the symbol "partner" to represent the concept "buyer or seller" (cf. Figure 2-12).

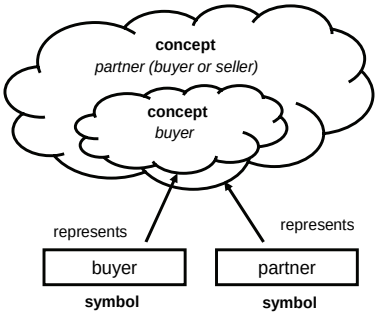


Figure 2-12
Different symbols, more general (specific) concepts

Besides differences in representing a single concept, sometimes there are differences in representing the relationships between two or more concepts.

Problem IP7. Different definition of the same concept (also known as confounding conflicts)

As said earlier a concept can be defined using already defined concepts. For example, if we have a concept of "person", "gender" and "male" we can define new concepts such as "man" (a person with male gender). Knowing the concept "parent", we can define the concept "father" (a parent and a man). Confounding semantic problems arise when concepts that abstract the *same real-world things* are *defined differently*. For example, one system may define "employee" as "a person who works for a company", whereas other system may define the same concept as "a person who is paid by a company" (cf. Figure 2-13).

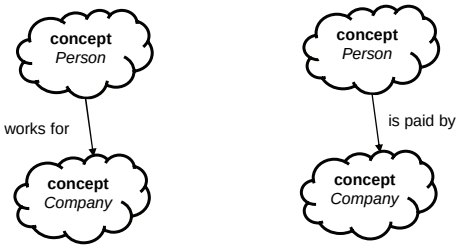


Figure 2-13
Confounding semantic problems

The awareness of the presented semantic mismatches is required to understand what semantic problems can arise when integrating different systems. In turn, this enables the systematic approach for resolving the problems which leads to building of interoperable integrated systems.

2.4 Pragmatic Interoperability

Systems interact by exchanging messages that contain data about some entity in the real world. When a system receives message *it changes its state, sends message back, or both* (Wieringa, 2003). In most cases, messages sent to the system change or request the system state, and messages sent from the system change or request the state of the system's environment[2].

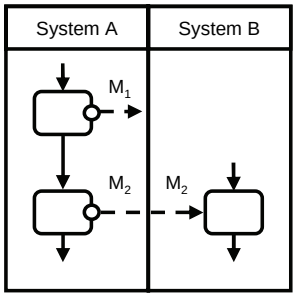
Pragmatic interoperability is concerned with ensuring that a message sent by a system causes the *effect* intended by that system. This means that a number of systems are pragmatically interoperable when they share the same expectations about of the effect of the messages they exchange. Often, an effect is achieved by sending and receiving multiple messages in specific order, defined in an *interaction protocol*.

Pragmatic interoperability problems arise when there are differences in the meaning the *data* in the exchanged messages (e.g., semantic problems) or there are differences in the *interaction protocols* of the systems that exchange these messages. We have presented the most common semantic interoperability problems in Section 2.3. In this section, we present the most common mismatches in the interaction protocols.

Problem BP1: Unexpected message mismatches arise when one system tries to send a message to other system, but the other system does not expect this message.

For example, *System A* intends to send first message M_1 and then message M_2 to *System B* whereas *System B* expect only the message M_2 (cf. Figure 2-14).

Figure 2-14 Unexpected message mismatch

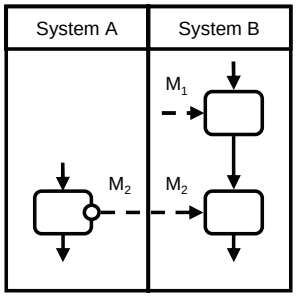


Problem BP2: Insufficient message mismatches arise when one system expects a message that is never sent by the other system.

[2] By environment of a system we mean all systems that are able to communicate with that system

For example, *System B* expects message M_1 but *System A* never sends message M_1 (cf. Figure 2-15). Consequently, a deadlock might occur.

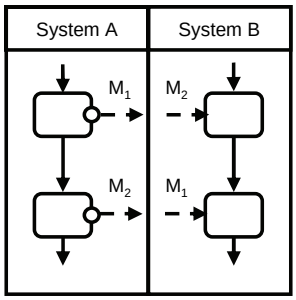
Figure 2-15
Insufficient message mismatch



Problem BP3: Message order mismatches arise when one system sends messages in a different order than expected by the other system.

For example, *System A* intends to send first message M_1 and then M_2 to *System B* whereas *System B* expects first the message M_2 and then M_1 (cf. Figure 2-16).

Figure 2-16
Message order mismatch



Problem BP4: Unexpected acknowledgement mismatches arise when one system sends a message to acknowledge the receiving of another message but the other system does not expect such an acknowledgement.

For example, *System B* receives a message M_1 and intends to send message Ack (e.g., to acknowledge the receiving of M_1) whereas *System A* does not expect such a message (cf. Figure 2-17).

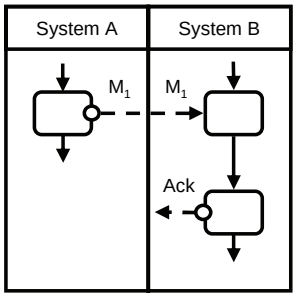


Figure 2-17 Unexpected acknowledgement mismatch

Problem BP5: Insufficient acknowledgement mismatches arise when one system expects an acknowledgement for receiving a message but the other system does not send such an acknowledgement.

For example, *System A* sends message M_1 and then expects a message Ack (e.g., an acknowledgement for the receiving of the message M_1), whereas *System B* does not intend to send such a message (cf. Figure 2-18). Consequently, a deadlock might occur.

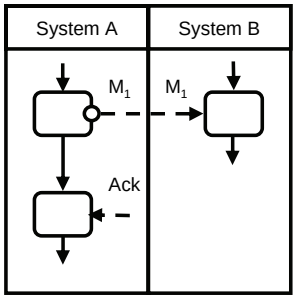


Figure 2-18 Insufficient acknowledgement mismatch

Problem BP6: Message aggregation mismatches arise when one system sends separate messages containing the same data that the other system expects in a single message.

For example, *System A* intends to send message M_1 and then M_2 whereas *System B* expects only one message that aggregates the data in M_1 and M_2 (cf. Figure 2-19).

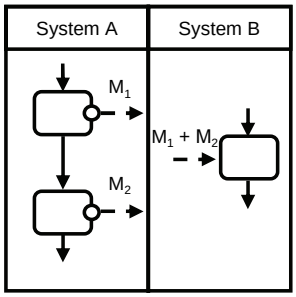


Figure 2-19 Message aggregation mismatch

Problem BP7: A more specific case of this mismatch is when one system sends a number of messages of type m *but the other system expects one message that contains a collection of all the messages (message* M*) (cf. Figure 2-20).*

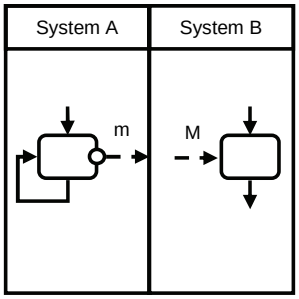


Figure 2-20 Message aggregation mismatch (collection of messages)

Problem BP8: Message splitting mismatches arise when one system sends a message containing the same data that the other system expects in multiple separate messages.

For example, *System A* intends to send one message with some data whereas the *System B* expects the same data in two separate messages (M_1 and M_2) (cf. Figure 2-21).

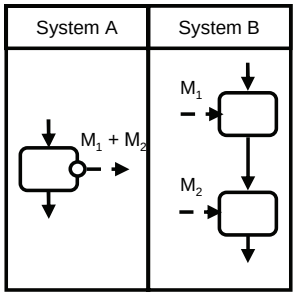


Figure 2-21 Message splitting mismatch

Problem BP9: A more specific case of this mismatch is when one system sends one message M that contains a collection of messages m but the other system expects separate messages m (cf. Figure 2-22).

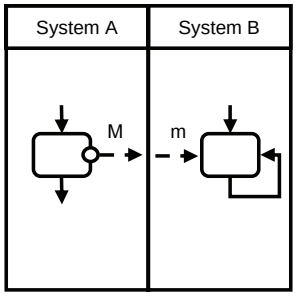


Figure 2-22 Message splitting mismatch (collection of messages)

2.5 Conclusions

In this chapter, we answered the *Research question Q1: "*What does interoperability mean? What does it mean for different systems to interoperate? Are there different levels of interoperability? What interoperability problems exist?*"*.

There have been many attempts to define what interoperability means. We have studied existing definitions and identified the common characteristics of interoperability. First, interoperability *involves multiple systems*, that is, we cannot talk about interoperability of one system. Second, interoperability is the *ability* of multiple systems *to interact*. Further, this interaction *requires little or no knowledge of the internal implementation* of the system. Finally, interoperability is about *achieving some common goal*. Based on the identified properties, we have defined interoperability in the context of SOA, namely as "*the ability of multiple systems to use each other's services effectively*".

Systems interact (i.e., they use each other services) by the means of a language. Using the distinction given by (Morris, 1938) we define three levels of interoperability, respectively *syntactic*, *semantic* and *pragmatic interoperability*. Syntactic interoperability is outside the scope of this thesis. For that reason, we focus only on semantic and pragmatic interoperability, defining what it means and what problems arise at these levels.

Semantic interoperability is concerned with ensuring that a symbol has the *same meaning* for all systems that use this symbol in their languages. Symbols are real world entities indirectly (i.e., through the concept they represent). Therefore, the semantic interoperability problems are caused either by

different abstraction of the same real-world entities or by *different representations of the same concepts*. In this chapter, we presented the most common semantic interoperability problems.

Pragmatic interoperability is concerned with ensuring that the exchanged messages cause their *intended effect*. Often, the intended effect is achieved by sending and receiving multiple messages in specific order, defined in an interaction protocol. Pragmatic interoperability problems arise when there are *differences in the meaning of data* in the exchanged messages (e.g., semantic problems) or there are *differences in the interaction protocols* of the systems that exchange these messages. In this chapter, we presented the most common differences in the interaction protocols.

Awareness of the possible interoperability problems enables system integrators to make more informed and carefully thought-out design decisions. In addition, the presented problem classification serves as an input when designing our service integration method. In Chapter 5, we analyse the problems presented in this chapter and provide solution for each of these problems.

Chapter 3

State-of-the-Art

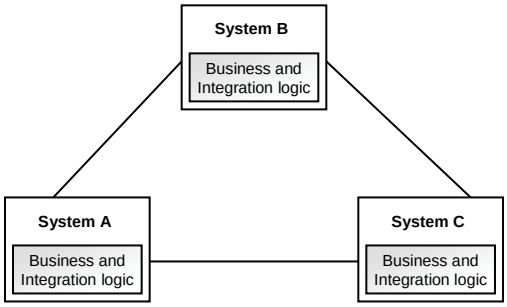
Enterprise Application Integration (EAI) is an extremely complex process. The reason is that the systems that have to be integrated have not been designed to work together, i.e., they are *heterogeneous*, *autonomous* and *distributed* (*HAD*). *Heterogeneous* systems use *different information models* to capture the semantics of the business domain. *Autonomous* systems exchange data following their own *interaction protocols independently* from the interaction protocols of any other system. *Distributed* systems *do not share common* state and use *different means* to *update* or *retrieve* this *state*. EAI is about *enabling* such HAD systems to interoperate.

In Chapter 1, we briefly introduced the EAI problem and the proposed solutions to deal with it. In this chapter, we present a short history of the EAI approaches, discuss their shortcomings, and argue what is required to address these shortcomings. The chapter is organised as follows: In Section 3.1, we present the most prominent EAI approaches and identify three main aspects of the EAI problem. The first aspect concerns the difference in the *information models* of the systems that have to be integrated. The second aspect concerns the difference in the *interaction protocols* of the systems. Finally, the third aspect concerns the *complexity of building* EAI solutions. In Sections 3.2, 3.3 and 3.4 we present *Service-Oriented Architecture (SOA)*, *Knowledge Representation (KR),* and *Model-Driven Architecture (MDA)*, respectively, as approaches to deal with each problem aspect. Finally, in Section 3.5, we argue that, since the problem aspects of current EAI approaches always occur together, SOA, KR, and MDA should be *combined* to deal with the problem as a whole.

3.1 Enterprise Application Integration (EAI) Approaches

EAI has developed over time in different phases. At the beginning, enterprises had to implement integration solutions themselves (so called *homegrown integration* (Busler, 2003)). The reason was that there were no integration products available on the market at that time. Enterprises started integrating their systems by *modifying* their *systems* in two different ways (cf. Figure 3-23):

Figure 3-23 Homegrown integration



- The systems have been modified to *call each other synchronously* and exchange data at the right moment of processing.

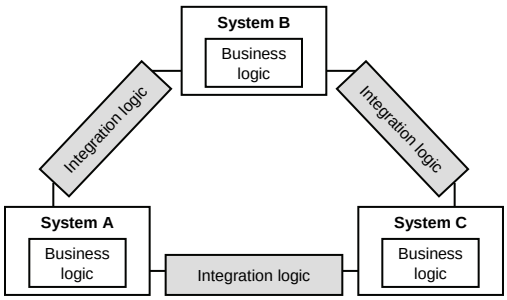
- The systems have been modified to *use an intermediate storage*, such as a file system or a dedicated database, to store and retrieve data that needed to be exchanged.

Over time, it became clear that these approaches have two main disadvantages. First, the system sending the data had to know about the system receiving the data. This means, that every time when a new system had to be added to or removed from the integration, the system sending the data had to be modified. Second, since systems have not been designed to interoperate, they had to implement data transformation logic themselves. For example, either the system that sends the data (or stores it at some location) had to transform the data in a format expected by the recipient or the recipient had to transform the data to its format. In this way, the systems had not only to implement the business logic but also to implement and mange the integration logic themselves. Ultimately, such tightly-coupled integration solutions, with no clear separation between business and integration logic became very difficult and expensive to maintain. This created market opportunity for integration products that did not require enterprise information systems to be aware of the integration. In the following, we present the most prominent integration approaches.

3.1.1 Point-to-Point Integration

In the *point-to-point* (P2P (Bussler, 2003)), a direct connection has been established between each pair of systems that needed to be integrated (cf. Figure 3-24).

Figure 3-24
P2P integration



The software, implementing the connection, is responsible for extracting data from the first system, transporting them and inserting data into the second system. When necessary, the integration software performs all required data transformations before inserting the data into the recipient system.

While providing basic integration functionality, the P2P approaches have some limitations that are unacceptable in more complex integration scenarios. First, for each new system that needs to be integrated, a new connection has to be added to each existing system part of the integration. In addition, logic that transforms data from (to) the new system and existing systems has to be specified. Second, more complex data exchange sequences cannot be defined. The reason for that is that connections between the systems are unaware of each other. For example, it is not possible to extract data from two different systems, combine it and then insert it into third system.

3.1.2 Hub-and-spoke

In the *hub-and-spoke* approach (Bussler, 2003), the communication between different systems has been implemented in a central system, called *hub*. The hub is responsible for receiving data from every system (called *spoke*), transforming it in the right format of a recipient system, and then inserting it into it (cf. Figure 3-25).

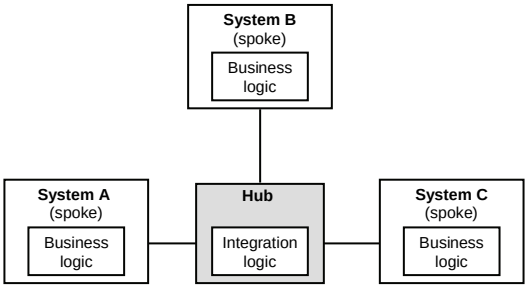


Figure 3-25 Hub-and-spoke integration

With this approach, the effort to maintain several different connections between enterprise information systems has been notably reduced.

Using a *publish/subscribe* mechanism, each spoke can provide its requirements to receive specific data. The hub matches the received data against these requirements, identifies the right spoke, transforms the data in the right format, and inserts it into the recipient system. Using publish/subscribe mechanism data from one system can be inserted into multiple destination systems.

The main advantage of hub-and-spoke approach is that adding a new system only requires adding one new connection between the system and the hub. The other systems, that have been already integrated, are not affected by the addition. In addition, new routing rules can be dynamically added to the hub enabling data to be correctly routed to the new system.

While improving on P2P integration approach, hub-and-spoke has some (major) drawbacks. First, it is not possible to implement *multistep integration*. For example, the following scenario cannot be achieved by P2P integration approach: a system sends a message to another system; the second system returns a message that has to be forwarded to a third system based on the content of the first message. The reason is that the data in the first message is no longer available. Second, hub-and-spoke provides only a *one-way integration*. For example, if a system sends a message to request date from another system and the second system responds, the hub does not know that the two messages are related. In this way, implementing even a simple request/response pattern requires definition of complex routing rules. Finally, the hub-and-spoke approach did not allow for adding additional *business logic* (such as notification or authorization) between the extraction and inserting of the data.

3.1.3 Process-based Integration

To address the limitations of hub-and-spoke integration approach, the *process-based* integration approach (Bussler, 2003) has been proposed. This approach extends the hub-and-spoke by adding process management

functionalities using a workflow management system (WFMS) (cf. Figure 3-26).

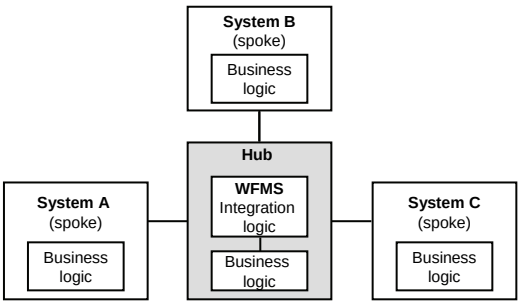


Figure 3-26 Process-based integration

In this way, instead of directly inserting the data into a recipient system, the hub inserts data into a workflow instance that determines the right way to process the message itself, together with other related messages.

The process-based integration approach supports both multistep integration and addition of custom steps between the extraction/insertion of the data. In fact, the process-based integration solutions are stateful, i.e., each instance of an workflow keeps all received and sent messages and can use this information to construct new messages or execute some processing logic. In this way, more complex business scenarios can be supported. In addition, the integration logic can be specified in a workflow definition, using constructs such as a conditional branching or a parallel execution.

Service-orientation is a promising design paradigm for building process-based integration solution. According to this paradigm different system has knowledge only about how to request and consume services provided by other systems, and has little (or no) knowledge about their internal data structures and processing logic. In this way, integration solutions can be specified using only service description (i.e., without knowledge about the internal implementation of these services) at higher level of abstraction.

3.2 Service-oriented Architecture

Service-oriented architecture (SOA) is an architecture comprised of services and service compositions designed and built in accordance with the service orientation paradigm. According to (Erl, 2005) SOA should adhere to the following principles:

– *Loose coupling* – systems maintain relationships that minimise dependences among them and only require that they retain an awareness

of each other. That is, when two systems need to be connected, instead of connecting them directly, a third system is used to mediate the connection between them. Loose coupling may have many dimensions such as loose coupling in time, location and version.

- *Contracts* – systems interact adhering to an agreement specified in a service description. This means that having only a service description (a contract) both service provider and service consumer should have everything they need to interact.

- *Autonomy* – systems have full control over their underlying runtime execution environment. That is, the only possible interaction with the system is through the services it provides. In this way, it is possible to change the implementation of the system without any effect on the consumers of its services.

- *Abstraction* - service descriptions should only contain essential information required to consume the service. A service description should not reveal low-level details about internal system's state and processing logic. However, finding the "right" level of abstraction is extremely difficult problem. For that reason, in Chapter 4, we define a conceptual framework that allows a service to be described at different levels of abstraction. In this way, different models of the same service can be used for different purposes (e.g., for discovery, composition and execution).

- *Reusability* – functions of a system should be exposed as services with the intention of promoting reuse. To achieve this, services should be independent units of logic. More complex business processes should be broken down into series of services, each responsible for performing independent portions of the business process.

- *Discoverability* – service descriptions should contain sufficient information to enable service users to discover, assess and use the service. Examples of such information are message formats, service endpoint and binding information but also organizational units and service-level agreements.

- *Composability* – a number of systems can be coordinated or new (compound) systems can be built from existing ones based entirely on their service descriptions. Similar to the discoverability principle, a service description should contain sufficient information to use a system as a component of another system.

– *Statelessness* – a service should minimise retaining information specific to a particular activity. Statelessness is preferred condition for services because it promotes reusability and scalability. It can be achieved by adding more intelligence to the exchanged messages, i.e., each message should convey all the necessary information for its processing.

To realise the principles of SOA a lot of effort is currently being invested in standardizing XML[3]-based service description languages and protocols for service interactions known as Web services. In the remainder of this section, we will briefly present the most prominent ones. Since there are many standards addressing different aspects of Web services such as WS-Coordination[4], WS-Policy[5] and WS-Trust[6], we limit ourselves only to the standards that are relevant to the research presented in this thesis.

3.2.1 Simple Object Access Protocol

The *Simple Object Access Protocol* (SOAP[7]) is a protocol for exchanging structured data. It uses XML as a data format and relies on other standards (such as HTTP[8]) for the actual transmission of the data. SOAP provides a basic messaging framework that is considered as foundation for building Web services. A SOAP message is a XML document that consists of a body and a header. The body contains the actual data that is used by the message receiver. The header provides support for advanced message processing. The information in the header is typically used by intermediate message processors. SOAP is independent from programming language and operational platform. I.e., it does not require a specific implementation technology which makes it agnostic to vendors, platforms, and technologies.

3.2.2 Web Services Description Language

Web Services Description Language (WSDL[9]) provides a model and format to describe a Web service. It allows service providers to describe both the abstract functionality of their services as well as to provide concrete details where to access a service and how. More specifically a WSDL description consists of two parts. The abstract part describes a Web service in terms of the messages it sends and receives through a type system. The cardinality and the sequence of the messages are defined by Message Exchange Patterns

[3] http://www.w3.org/XML/
[4] http://www.ibm.com/developerworks/library/specification/ws-tx/
[5] http://www.w3.org/Submission/WS-Policy/
[6] http://docs.oasis-open.org/ws-sx/ws-trust/
[7] http://www.w3.org/TR/soap/
[8] http://www.w3.org/Protocols/Specs.html
[9] http://www.w3.org/TR/wsdl

(MEPs). An operation associates a MEP to one more messages. Finally, a number of operations are group into an interface. The concrete part describes the implementation details necessary to access the Web service. A binding specifies the concrete message format (e.g., SOAP) and transmission protocol (e.g., HTTP). A service endpoint associates network address with a binding. Finally, a service groups the endpoints that implement a common interface.

3.2.3 Universal Description, Discovery and Integration

Universal Description, Discovery and Integration (UDDI[10]) is a standard that defines a standard way of registering and looking up services. In a typical UDDI scenario, a service provider registers a service in a UDDI registry. Then, a service consumer can look up in the registry for a required service. When the consumer request matches a service offer, a service description is returned. The consumer uses the returned service description to bind directly with the service provider and use the service.

3.2.4 Web Services Business Process Execution Language

The *Web Services Business Process Execution Language* (WS-BPEL[11]) is a standard for describing service-oriented processes, i.e., processes in which each action is performed by a Web service (including sending and receiving of a message). WS-BPEL provides support for specification of both service orchestration and choreography (from the single partner perspective). A WS-BPEL orchestration specifies the internal behaviour of a composite Web service in terms of its components (i.e., other Web services) and the relations between their operations. A WS-BPEL choreography specifies the external behaviour of a Web service by specifying the relations between the service operations.

The orchestration can be seen as the *private business process* of the service provider. It is controlled by the service provider and describes the steps of its internal, executable workflow. The choreography can be seen as the *public business process* from the perspective of the service provider. It is describes the sequence of externally observable messages between the service and its users.

A WS-BPEL process usually involves participation of different *partners*. They interact through interfaces called *port types*. Port types are related by *partner links*. A partner link specifies what port types must be supported by each of the partners it connects. A partner link is an instance of a *partner link*

[10] http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm
[11] http://www.oasis-open.org/apps/org/workgroup/wsbpel/

type which defines the *roles* of each partner. A WS-BPEL process exchanges *messages* with its partners. These messages are defined in WSDL.

A WS-BPEL process uses *variables* to hold the data in the messages, exchanged between the process and its partners, or internal process data. A *variable* has a type, defined in the *types* section of a WSDL document, a type of an element defined in apart XML Schema or XML Schema simple type.

A WS-BPEL process consists of activities and relations between them. The activities can be split into two groups – *basic* activities and *structured* activities. Examples of basic activities are *receive* (used to block the execution of the process and to wait for a matching message to arrive), *reply (*allows the process to send a message in reply to a message that was received by an *receive* activity) and *invoke* (allows the process to invoke *one-way* or *request-response* operation on a Web service offered by a partner). Examples of structured activities are *sequence* (used to specify the execution of activities in a sequence), *if* (used to specify a conditional behaviour), *while* (used to specify a repetitive behaviour), *flow* (used to specify one or more activities to be performed in parallel) and *pick* (used to block the process and wait for the occurrence of exactly one event, specified in a set of events). For the complete list of WS-BPEL activities and their detailed description, we refer to the WS-BPEL standard.

Web service standards only standardise the *syntax* of service descriptions. They do not provide means for defining their *semantics*. This means that although syntactically correct, a service description still can be ambiguous and therefore, misunderstood by its intended users. Knowledge representation approaches enable the explicit and precise specification of service semantics. In the next section, we present an overview of the most prominent knowledge representation approaches. More specifically, we focus on ontology representation languages.

3.3 Ontology Representation

The term "ontology" has been accepted by the IT community to describe formal domain models. There are several ontology definitions. The most accepted one is given in (Gruber, 1993): "an ontology an explicit specification of a conceptualization". (Borst, 1997) defines ontology as "formal specification of a shared conceptualization". In addition to the definition given by Gruber, Borst requires that the conceptualization should be shared between several parties and specified in a formal way. (Studer, 1998) combines the definitions of Gruber and Borst into "an ontology is a formal, explicit specification of a shared conceptualization". The shared conceptualization captured in an ontology enhances the communication between humans. The formal specification of this shared conceptualization

enhances the communication between machines. For these reasons, ontologies are promising approach to automate information integration tasks. In this thesis, we will use the ontology definition given by Studer.

There is no universally accepted language to represent ontologies. Nowadays, there are many language specifications, graphical and natural language notations to represent ontologies. However, when deciding on which ontology representation language to use, some basic criteria should be considered (Pollock and Hodgson, 2004):

- *Processability* – the ability of the representation language to be efficiently processed by software
- *Accessibility* – the market penetration and familiarity of the representation language in the industry and among professional that are going to use it
- *Usability* – the ease with which new users can learn and use the representation language
- *Expressiveness* – the ability of the language to capture unambiguously the semantics of the subject domain of a given systems
- *Life cycle coverage* – the scope of the representation language throughout the development cycle (e.g., design, validation, implementation, testing)

In the rest of this section, we briefly present the most prominent candidate languages for ontology representation and discuss their strengths and weaknesses.

3.3.1 Resource Description Framework

Resource Description Framework (RDF[12]) is a language for representing information about resources on the Web. Initially intended for representing metadata about Web resources it has been generalised to represent also information about entities (or phenomena) in the real world.

The basic RDF data model consists of three object types:

- *Resources*: All things described by RDF are called resources. A resource may be an entire web page, part of a web page or a collection of web pages. A resource may also be an object that is not directly accessible via

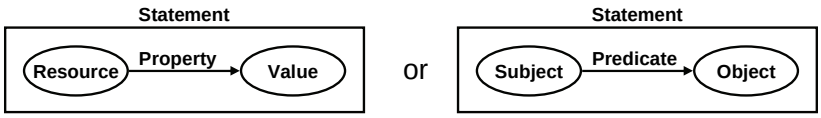
[12] http://www.w3.org/RDF/

the web (e.g., a physical entity). All resources in RDF are identified by *Uniform Resource Identifiers* (URIs[13]).

– *Properties*: A property is a specific aspect, characteristic, attribute, or relation used to describe a resource. Each property has a specific meaning, defines its permitted values, the types of resources it can describe, and its relationship with other properties.

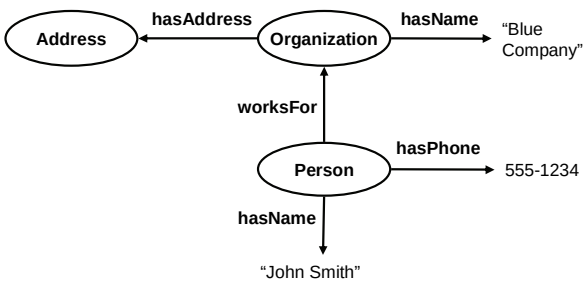
– *Statements*: A statement is a specific resource together with a named property and the value of that property. These three individual parts are called, respectively, the *subject*, the *predicate*, and the *object* of the statement. The object of a statement (i.e., the value of the property) can be another *resource* or a *literal* (e.g., a primitive data type defined by XML Schema). In RDF terms, a literal may have content that is XML mark-up but is no further evaluated by the RDF processor. The underlying structure of any expression in RDF can be viewed as a directed labelled graph, which consists of nodes and labelled directed arcs that link pairs of nodes, as depicted in Figure 3-27.

Figure 3-27
RDF statement



The RDF graph is a set of statements. The direction of the arc is significant: it always points toward the object of a statement. The meaning of an RDF graph is the conjunction (i.e., logical and) of all statements that it contains. Figure 3-28 shows an example of an RDF graph.

Figure 3-28
RDF graph



[13] http://www.w3.org/Addressing/

Data types are used by RDF in the representation of values such as integers, floating point numbers and dates. RDF uses the data types defined by XML Schema[14]. It does not provide mechanism for defining new data types. XML Schema data types provide an extensibility framework suitable for defining new data types for use in RDF.

An RDF graph, as described by the RDF abstract syntax, can be represented in various ways, using different concrete syntaxes but each conveying a common RDF meaning. Only the XML syntax is normatively specified and recommended for use to exchange information between applications.

Resource description communities require the ability to state certain things about certain kinds of resources. For describing bibliographic resources, for example, descriptive attributes including "author", "title", and "subject" are common. For describing business entities, attributes such as "partner" and "purchase order" are required. In RDF, the declaration of these properties (attributes) and their corresponding semantics are defined by RDF Schema[15].

RDF Schema is specified in terms of the basic RDF information model - a graph structure describing resources and properties. All RDF vocabularies share some basic common structure: they describe classes of resource and types of relationships between resources. This commonality allows for a finer grained mixing of machine-processable vocabularies, and addresses the need to create metadata in which statements can draw upon multiple vocabularies that are managed in a decentralised fashion by independent communities (W3C, 2004c).

The RDF Schema approach to vocabulary description allows vocabulary designers to represent descriptions of classes and properties, for example by describing ways in which combinations of classes, properties and values can be used together meaningfully.

The example depicted in Figure 3-29, illustrates the way in which RDF can be used to describe real world things (e.g., *people* and *companies*), the classes they fall into (such as *Employee*), and the properties that are used to relate members of these classes - in this example the property *worksFor*. By using a RDF Schema, we can describe the relationship between RDF properties and these classes of resource. In this example, the RDF Schema is used to say that the *worksFor* property relates *Employees* to *Companies*. The example also shows that all *Employees* are considered to be *Persons*.

[14] http://www.w3.org/XML/Schema
[15] http://www.w3.org/TR/rdf-schema/

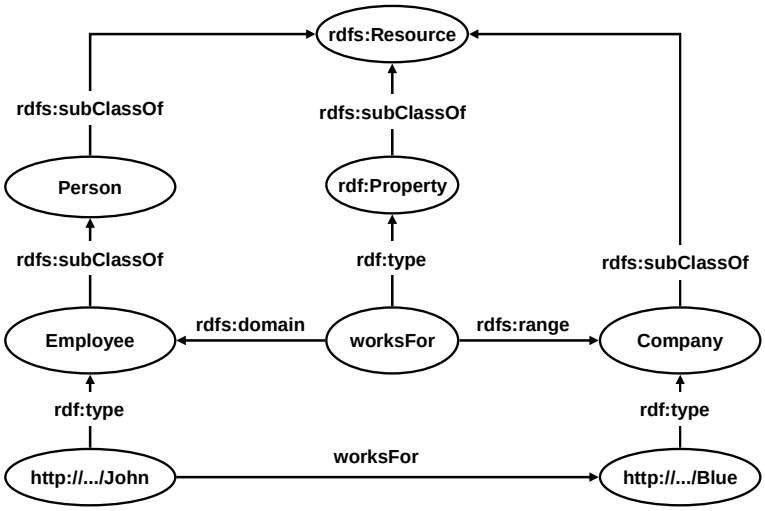


Figure 3-29
An example of RDF Schema instance

RDF Schema consists of a collection of RDF resources that can be used to describe properties of other RDF resources (including properties) which define application-specific RDF vocabularies. The core vocabulary is defined in the namespace "http://www.w3.org/2000/01/rdf-schema#".

The RDF Schema class and property system is similar to the type systems of object-oriented programming languages such as Java. However, RDF differs from many such systems in that instead of defining a class in terms of its properties, an RDF schema will define properties in terms of the classes to which they apply. For example, we could define the *worksFor* property to have *domain* of *Emploee* and a *range* of *Company*, whereas a classical object-oriented system might typically define a class *Person* with an attribute called *worksFor* of type *Company*. This example is depicted in Figure 3-30.

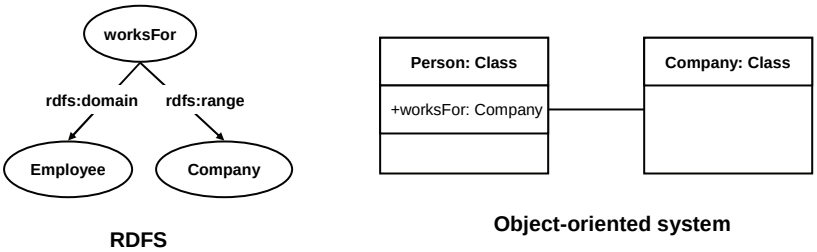


Figure 3-30
Example of RDF Schema and alternative OO Class diagram

By using the RDF approach, it is easy for others to define subsequently additional properties with a domain Person or a range Company. This can be done without the need to re-define the original description of these

classes. One benefit of the RDF property-centric approach is that it is very easy for anyone to say anything they want about existing resources, which is one of the architectural principles of the Web (Berners-Lee, 1998).

RDF Schema provides a mechanism for describing information, but does not say whether or how an application should use it. Different applications will use this information in different ways. For example, a data-checking tool might use RDFS to discover errors in some dataset, an interactive editor might suggest appropriate values, and a reasoning application might use it to infer additional information from instance data.

RDF Schemas can describe relationships between vocabulary items from multiple independently developed schemas. Since URI references are used to identify classes and properties in the Web, it is possible to create new properties that have a domain or range whose value is a class defined in another namespace. This makes RDF suitable as a language to represent mappings between different resources.

3.3.2 Web Ontology Language

Web Ontology Language (OWL[16] (Dean, 2004)) is a World Wide Web consortium's (W3C[17]) standard for representing ontologies. OWL builds upon the RDF and RDF Schema. In this sub-section, we present OWL semantics and discuss the supported reasoning tasks. We only focus on the OWL semantics.

In OWL, *individuals* are RDFS resources and represent entities (or phenomena) in some domain of interest. Logically, an individual is an assertion of existence, e.g., by defining the individual John we assert the existence of (a person) John in the real world.

Two individuals can be asserted to be the *same* or *different*. For example, if we assert that the individuals *John* and *Johnny* are the same, we mean that they both refer to the same entity in the real work, e.g., (the person) John.

Individuals can have *properties* and *type*. For example, we can state that *John* is a *Person*.

In OWL, *classes* provide an abstraction mechanism for grouping individuals with similar properties, for example, *Persons*, *Cities*, *Countries* or *Cars*. A class has an *intentional* meaning, i.e., it is a *membership criterion* or equivalently - a *unary predicate*. In OWL, two special classes *Thing* and *Nothing* are defined by the membership criteria always *true* and always *false*, respectively. This means that all individuals are members of the class *Thing* and no individual is a member of the class *Nothing*.

The membership criterion point of view allows us to identify a class with a unary predicate. For some classes this membership criterion is explicit

[16] http://www.w3.org/TR/owl-ref/
[17] http://www.w3.org/

(for example, having a red colour), but often that membership criterion is axiomatic (e.g., by stating that the individual *John* is a *Person*). Every class is associated with a set of individuals, called the class extension. For these individuals it is asserted implicitly or explicitly that they satisfy the class membership criterion. In particular, such an assertion claims the existence of such individuals. Two classes may have the same class extension, but still be different classes. For example, we often use classes for which we know no members at all. It is often convenient to think of a class as a set with some known but, potentially many more (unknown) individuals characterised by some property.

A *property* is a binary relationship (i.e., a *binary predicate*) from one class to another or from a class to a data type. We can assert that two individuals or an individual and a data value are related through a property, for example, *worksFor(John, TelematicaInstituut)* or *hasAge(John, 35)*.

A property usually has a *domain* and *range* class. If not defined explicitly, a property is assumed to have domain and range the class *Thing*. A useful way to think about the domain and range is that *P* is a multi-valued function defined on its domain with values in its range. If class *A* is the domain of property *P,* and class *B* is the range of *P*, then every time when we explicitly assert a triple $x\ P\ y$ we implicitly assert that $x \in A$ and $y \in B$.

For example if the class *Person* is the domain, and the class *Organisation* the range of the property *worksFor*, we conclude from the existence of the fact *worksFor(John, TelematicaInstituut)* that $John \in Person$ and $TelematicaInstituut \in Organisation$.

The properties are defined independently of the triples, just as classes are defined independently of their members. A property is a binary predicate and just like classes (unary predicates) has an intentional meaning. A triple is an assertion that (subject, object) pair satisfies the binary predicate corresponding to that property. Likewise, the assertion that an individual is a member of a class is the assertion that the individual satisfies the unary predicate corresponding to that class. Thus, the set of all triples with a given property as predicate is completely analogous to the extension of the class.

If we state the existence of individuals, properties or classes we do not make a claim that these individuals, properties or classes are the only ones that exist or could exist. We merely state that these are known and have to be considered in the interpretation of the ontology they belong to. Such an assumption is particularly suited for an environment like the Web where we have to assume incomplete knowledge. However, in many applications the situation is opposite. For example, if we have a database, then the database records are all the individuals in the database, and a table's columns define all its properties. Moreover, the result of a query is an authorative answer whether the returned records do or do not satisfy the query. These different

points of view are complementary. A database often contains incomplete information, and the answer of a query is merely authorative for information that has been stored in the database. Conversely, we can state that a class consists exactly of a number of instances and no others.

OWL classes can be defined in different ways:

- *axiomatically*, by stating that the class exists, for example, the class Human
- *extensionally*, by enumerating all individuals that belong to the class, for example, the class {*Belgium*, *Netherlands*, *Luxembourg*}
- *intentionally*, by defining the membership criteria of the class, for example, having a red colour.
- *as the intersection* of two or more classes, for example, Human $\cap$ Male. In terms of membership criteria, this is the conjunction of the criteria of the classes, e.g., Human and Male.
- *as the union* of two or more classes, for example, American $\cup$ Canadian. In terms of membership criteria, this is disjunction of the criteria of the classes, e.g., American or Canadian.
- *as the complement* of another class, for example, $\neg$Vegetarian. In terms of membership criterion this is negation of the original criterion, e.g., not Vegetarian.

Classes can be organised into a *subclass-superclass hierarchy* (i.e., taxonomy). In terms of the membership criterion, a class *C* is a subclass of a class *D* if the membership criterion for *C* implies the membership criterion for *D*. In OWL the class *Nothing* is a subclass of all classes and no individual is a member of *Nothing*. Likewise, *Thing* is a superclass of all classes and all individuals are member of *Thing*.

We usually just assert that one class is a subclass of another. For example, we can define *Father* as a subclass of *Man* and *Man* as a subclass of *Human*. An individual that is a member of a class is also a member of all its superclasses.

Two classes are *equivalent* if they are subclasses of each other. Thus, a necessary and sufficient condition to belong to a class is to belong to an equivalent class.

Classes can be asserted to be *disjoint*. This means that the conjunction (the logical and) of the corresponding membership criterion is always false, which means that the classes cannot have common members. For example,

if we define Man and Woman to be disjoint classes, no individual can be member of Man and Woman at the same time. The definition implies in particularly that the members of the two disjoint classes are distinct.

A class can also be defined by a *restriction* on property values. An *existential* (∃) restriction (from the existential quantifier ∃, which reads as "for some," "there exists," or "for at least one") has as membership criterion that for some given property *P* there exists a value instance in some given class *C*.

OWL provides useful variations of the existential restriction. The simplest is the *hasValue* restriction which asserts the existence of a specific property value. For example, Londoner is someone who *livesIn* the city London.

Another variation is the *cardinality* restriction (or more precisely *qualified cardinality restriction*). In this case, a necessary and sufficient membership condition is that there is a more precisely defined number of property values to other individuals or data values.

Negating the existential condition, we are led to the *universal* (∀) restriction (coming from the universal quantifier ∀, which reads as "for all"). The membership criterion for the universal restriction is that all (also possibly zero) values from a property *P* are members of a class *C*.

Like classes, properties can be more or less specific which leads to property hierarchies. A property *R* is a subproperty of a property *P*, denoted $R \subseteq P$, if *R* implies *P*. It follows that each asserted fact *R(x, y)*, implies a fact *P(x, y)*. For example, if John has a daughter, then in particular, John has a child.

Some properties in OWL have special semantics. This allows an asserted fact to imply other facts. Examples of OWL properties with a special semantics are inverse, symmetric and transitive properties.

OWL is a set of three, increasingly complex languages:

- *OWL Lite* has been defined with the intention of creating a simple language that will satisfy users, primarily needing a classification hierarchy and simple constraint features. For example, OWL Lite supports cardinality constraints but only permits cardinality values of zero or one.

- *OWL DL* includes the complete OWL vocabulary, interpreted under a number of simple constraints. Primary among these is that a class cannot be a property or an individual simultaneously. Similarly, properties cannot be individuals. OWL DL is so named after its correspondence to Description Logics (Calvanese, 2003).

– *OWL Full* includes the complete OWL vocabulary, interpreted more broadly than in OWL DL, with the freedom provided by RDF. In OWL Full, a class can be treated simultaneously as a collection of individuals (the class extension) and as an individual in its own.

As we have seen, the formal definitions of classes, properties and individuals allow inferring new knowledge from knowledge that is already present. The basic inferences can be combined, and allow us to do more complex reasoning. It is useful to distinguish *property-*, *class-* and *individual-level reasoning*.

Property-level reasoning means inferring implied triples from the asserted ones. This is a closure process that constructs the implied (in the examples above - the triples with a dotted line) triples. For example, for a transitive property we have to create the transitive closure of the graph defined by the triples with the transitive property as predicate.

Class-level reasoning means checking whether a class B is a subclass of class A. This reasoning task is called a subsumption check. In other words, subsumption is checking if the criteria for being member of class B imply the criteria for being member of class A. If A and B subsume each other they are equivalent and in particular have the same members. Checking class satisfiability is a special case of subsumption reasoning. A class C is called unsatisfiable if $C \subseteq$ Nothing, hence (since Nothing $\subseteq C$ by definition), C is equivalent to Nothing, and cannot have any members. Conversely, we can check the subsumption $B \subseteq A$ by checking that the class $B \cap \neg A$ is unsatisfiable. If we construct the full subsumption graph of all classes in an ontology, we construct a class hierarchy. This reasoning task is called classification.

Individual-level reasoning means checking if an individual can exist in some model (called a consistency check) In particular if a class is unsatisfiable it cannot have a individual as a member. Thus, to check if the class C is satisfiable, it suffices to check that there is no model with a member $x \in C$.

A related task is to find the classes of which an individual is a member (called realization). Since we can construct a class hierarchy, we can in particular find the most specific class(es) to which an individual belongs. If we do this for only one class (which is a membership criterion) and find all the known instances that provably belong to the class we say we retrieve its instances. Instance retrieval is of great practical importance because it allows a logical model to be used as a database.

OWL has a number of strengths that make it very suitable for ontology representation. First, it is an effective union of object-oriented and logic-based systems. Second, OWL is *decidable.* This means that querying ontology representation always terminates, i.e., the query always has an answer. Third, OWL is very *expressive* language capable of describing real-world

entities and complex relationships among them. Finally, OWL has well-studied computational properties and is highly optimised for efficient processing by machines. However, OWL has also some weakness that should be considered before using it. First, it is relatively new standard with a little market penetration. Currently, there are only few tools available, most of them implemented in the academic world. Second, OWL is a logical formalism that requires different way of thinking than traditional data modelling approaches. This makes it unintuitive for people that have experience with object-oriented or relational modelling approaches.

3.3.3 Entity-Relationship Diagrams

Entity-relationship diagrams (ERDs (Chen, 1976)) are one of the most popular approaches for representing knowledge about the subject domain of a system.

An *entity* is a discrete thing that can exist independently from other things and can be uniquely identified. An entity can be a physical object, such as a person or a car, a non-physical object, such as a promise or an obligation, or an event such as selling a house or performing a task.

Each entity is an instance of a concept and can be referred to using that concept. In ERD, concepts are called *entity types* and graphically represented by a named rectangle. All instances of a type share certain properties. For example, a person has a name, a house has an owner, and a car has a colour. These properties are called *attributes*. An *attributes* can be *atomic* or *composed* of other attributes. In this way, more complex entity types can be defined. In addition, attributes may have *cardinalities* stating whether an attribute is optional or mandatory, or whether it is single- or multi-valued. An attribute is represented as an oval connected to the respective entity type (or composite attribute).

A *relationship* between entity types is a set of tuples of instances of these types. The number of different entity types that are connected by a relationship defines the *arity* of the relationship. A relationship is represented by a diamond and lines connecting the diamond and the respective entity-type rectangles. A relation has a *name,* which can be optionally labelled by an arrow representing the direction of the relationship. A *role* name can be added to each end of the line representing the role played by the entity in the relationship. In addition, *cardinality* can be added to each end of the line. Cardinality specifies how many instances of that one entity type can exists for each existing instances of some other entity type. Figure 3-31 illustrates an example ERD.

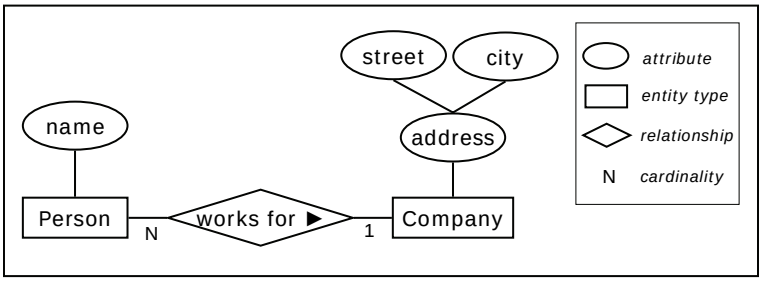


Figure 3-31
Example ERD

The example ERD describes a subject domain in which two types of entities can exist, namely, *Person* and *Company*. *Person* has attribute *name* and Company has attribute *address*. Address is a composite attribute: is consists of two sub-attributes, namely *street* and *address*. *Person* and *Company* can be related by the relationship *works for*. *One Person* works for *one Company* and *many Persons* can work for the *same Company*.

A relationship can also have attributes. In this case, the relationship is called *association entity*. This is represented by a diamond placed in a containing rectangle. The difference between a "normal" entity and an association entity is in the way we identify them. A "normal" entity has an *independent identity* whereas an association entity is identified by the entities that participate in the respective relationship. Figure 3-32 illustrates an example of an association entity.

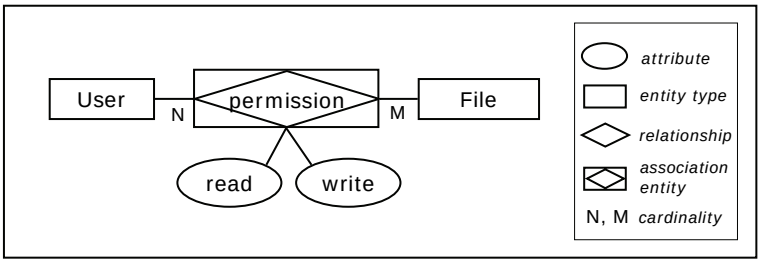


Figure 3-32
Example of an association entity

Chen's ERD does not support modelling organizing entity types into generalization/specialization hierarchies. To address this issue, (Elmasri and Navathe, 2000) have proposed *Enchanced Entity Relationship* (EER) diagrams. In EER, instead of using specialization and generalization, *subtype* and *supertype* are used. All instances of a subtype are also instances of all supertypes of that type. This also means that all instances of a subtype also have the properties of the respective supertypes. This is called inheritance.

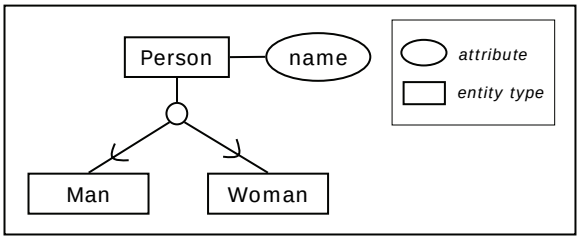


Figure 3-33 Example of a specification/generalization

In this example, *Person* is a supertype of the types *Man* and *Woman*. *Man* and *Woman* inherit all attributes of *Person*, i.e., they both have attribute *name*.

3.3.4 UML

Unified Modelling Language (UML[18]) is a general-purpose modelling language standardised by Object Management Group (OMG[19]). UML provides concepts and notation to model the structure and the behaviour of software systems.

Structural diagrams represent the decomposition of a system in terms of components, classes and objects as well as the relations among them. *Behaviour diagrams* represent what activities are performed by the system and how these activities change the state of the system. *Interaction diagrams* are subset of the behaviour diagrams. They are used to model the flow of control and data among the elements of the system.

Structural diagrams are similar to ERD, but they use different terminology. (Wieringa, 2003) presents a comparison between terminology used in UML structural diagrams and ERD. In table Figure 3-34, we summarise the most important part of the comparison.

Figure 3-34 Comparison of terminology in ERD and UML structural diagrams

ERD	UML structural diagrams
Entity type	Class
Entity	Object
Relationship	Association
Association entity	Association object
Association entity type	Association class
Cardinality property	Multiplicity property

The main difference between UML structural diagrams and ERDs is that ERDs are used to model the entities in the subject domain of a system whereas UML structural diagrams are used to model the software objects that make up the system.

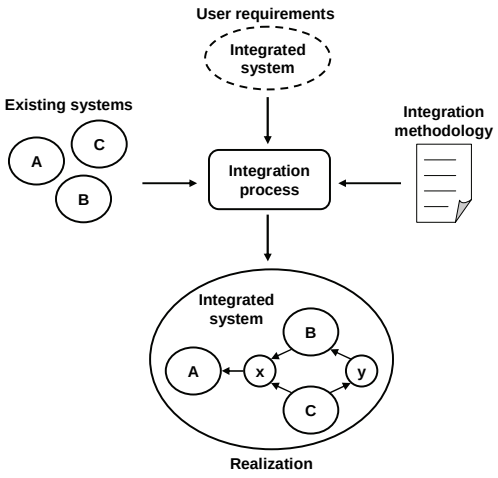
[18] http://www.uml.org/
[19] http://www.omg.org/

UML has wide market penetration. There are many, mature tools offered by different vendors. The main weakness of UML is that it is not grounded with a formal semantics. For that reason, in some cases models expressed in UML are ambiguous.

3.4 Model-Driven Architecture

System integration can be seen as the process of realizing an integrated system that satisfies some user requirements while using some integration methodology. The integrated system is built by linking existing systems and compensating the mismatches between them by adding additional systems called adaptors (Figure 3-35).

Figure 3-35
Integration process



Since the gap between user requirements and the realization of the integrated system can be wide, the integration process can become extremely complex. To simplify this process a good integration methodology should allow systems integrators to *address* only a *limited set of concerns* in series of design steps. This principle is known as *separation of concerns*.

To address the separation of concerns principle the OMG has proposed the *Model Driven Architecture* (MDA[20]) approach for software development. In MDA, the separation of concerns is achieved by specifying *models* at *different level of abstraction*. Each of these models focuses on the characteristics of an entity (or phenomenon) that are considered as being essential for a certain purpose, while ignoring or discarding details that are considered as being

[20] http://www.omg.org/mda/

irrelevant for the same purpose. In MDA, models can be automatically derived from other models by applying *transformation* activities.

MDA distinguishes four types of models: *computation-independent models (CIMs)*, *platform-independent model (PIMs)*, *platform-specific models (PSMs)* and *transformation models (TMs)*.

A CIM defines the business problem to be solved by a software system, e.g., the business goals to be achieved by the integrated system, its organizational structure and the associated business processes.

A PIM defines a technology-independent solution of the business problem as defined in a CIM. That is, a PIM is used to bridge the gap between the problem and solution spaces. Business experts should be able to review a PIM and check whether it captures the business problem as defined in the CIM. IT experts should be able to review a PIM and check if it matches their IT solution.

A PSM defines the realization of a system by means of a specific technology, for example J2EE[21] or .NET[22]. Note that often someone's PSM is someone else's PIM. For example, for a software architect a PIM can be specified in terms of the *Business Process Modelling Notation* (BPMN[23]) and transformed to a PSM in terms of Web services and WS-BPEL. However, for a Java developer the architect's PSM is a PIM that can be transformed to a PSM in terms of Java.

Finally, a TM defines how to transform elements from one model to another. For example, a TM could specify how a PIM in terms of Web services is realised by a specific set of implementation technology such as Java.

The relationships between PIM and PSM models are illustrated in Figure 3-36.

[21] http://java.sun.com/javaee/
[22] http://msdn.microsoft.com/netframework/
[23] http://www.bpmn.org/

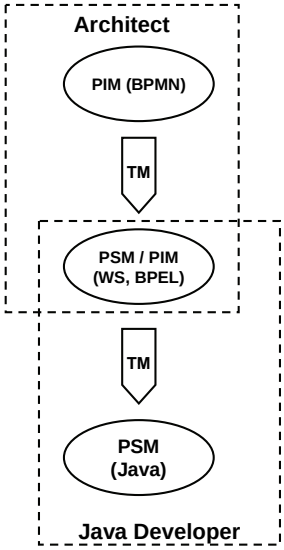


Figure 3-36
Example of the relations between PIM and PSM in MDA

In MDA, models are specified in well-defined languages, suitable for automated processing by machines. These languages, in turn are defined by models called *meta-models*. Since a meta-model is also a model, it must be written in some language (sometimes called *metalanguage*). In theory, there is an infinite number of layers of model-language-meta-model-metalanguage. The OMG defines four such layers, denoted as M0, M1, M2 and M3.

Models at layer M0 describe instances that exist in some system, i.e., the state of the system. For example, in a relational database management system an instance can be a particular record in that database. In a business system, an instance can be a concrete person or product.

Models at layer M1 define the kind of instances that can exist in layer M0, their properties, and the types of relations among them. For example, in case of a database system a model at layer M1 defines the tables "Customer" and "Order", their columns (e.g. "name", "address" and "orderId"), and the relationship between them.

Models at layer M2 define the kind of elements that can exist in M1, i.e., models of M2 are the meta-models of the models at layer M1. For example, in case of a database system a model at layer M2 defines the concepts "Table", "Column", "Primary key" and "Foreign key", as well as the relationships among them (e.g. a *Table* has at least one *Column*).

Finally, models at layer M3 define the kind of elements that can exist at layer M2, i.e., models of M3 are the meta-models of the models at layer M2. To define the models at layer M3, OMG has standardised a special

language called *Meta Object Facility* (MOF[24]). All modelling languages within MDA (e.g., UML) are instances of MOF.

In addition to MOF, the OMG has standardised a set of languages for formally defining the transformation among models. The standard is called *Query, Views and Transformations* (QVT[25]). The QVT specification has a hybrid declarative and imperative nature.

Meta-modelling and transformations in MDA enable *domain-specific modelling* (DSM) as well as definition of *domain-specific languages* (DSLs). DSLs are languages tailored to a specific domain as opposed to general purpose languages (GPLs) that are designed for any kind of application domain. Compared with GPLs (e.g. UML and Java), DSLs offer only a limited set of constructs. This increases the modelling or programming productivity and enables more precise definition of concerns within a particular domain.

The main advantages of using DSLs are discussed in (Mernik, 2006):

- A DSL has a syntax or graphical notation that is closer to the terminology or the notation used by domain experts. This makes it easier for them to learn the DSL and to use it to specify their requirements in more formal and precise way. This, in turn, narrows the gap between the problem and the solution spaces.

- A specification written in a GPL is difficult (or sometimes impossible) to analyse, verify and optimise because the GPL constructs are too complex or not well-defined. A DSL with formal reasoning capabilities enables solution designers to analyse, verify and optimise their models and to discover possible problems at early stage of the solution development process.

- Finally, software programmers often spend significant time on tasks that are tedious and follow the same pattern. In such cases, a single DSL construct can be used to generate all the required code in the respective GPL. This, in turn, improves the programmers' productivity.

It is important to mention that developing a good DSL is a very hard task. It requires very good understanding of the problem domain as well as very good tool support (e.g., parser generators and code generators). In addition, DSLs are only useful when the problem that has to be solved is

[24] http://www.omg.org/technology/documents/formal/mof.htm
[25] http://www.omg.org/docs/ptc/07-07-07.pdf

reoccurring. Otherwise, the cost to develop a DSL is higher than solving the problem with a general-purpose language.

3.5 Conclusions

In this chapter, we answered *Research question Q2*: "What are the current system integration approaches? What are their drawbacks? What technologies have been proposed to address these drawbacks? How do these technologies interact when used together?".

In Section 3.1, we presented a brief history of EAI. We showed how integration approaches have evolved addressing different issues of their predecessors. Nevertheless, there are still some main issues that have to be resolved.

First, to enable different systems to interoperate, system integrators need to know the *syntax* and the *semantics* of the data requested and provided by the systems. Only having this knowledge, they can specify the correct data mappings to deal with the mismatches in the information models of the different systems. Second, to achieve the goal of the integration, system integrators also need to know the *interaction protocols* (i.e., the correct order of data exchanges) of the systems that have to be integrated. Only having this knowledge, they can specify the integration logic required to deal with the mismatches in the interaction protocols of the different systems.

Third, the existing EAI approaches require system integrators to specify executable integration logic. I.e., all steps of the integration logic and all data transformations have to be fully specified in some language supported by the respective EAI product. Usually, these languages are too technical for business domain experts to understand and use. This limits their participation in the integration process to simply defining integration requirements in some informal way. E.g., current EAI approaches do not allow domain experts to specify an abstract solution nor to review the solution specification made by IT professionals. In addition, since the integration solution is specified for a concrete EAI technology it is very difficult to reuse it when the implementation technology changes.

As we can see, the EAI problem has three aspects that always occur together. Therefore, the solutions to each of them should always be combined. In this thesis, we propose SOA, KR and MDE to deal with different problem aspect of the current EAI. Figure 3-37 presents the relation of problem aspects and proposed solution approaches.

Problem aspect	Solution approach
Data aspect • Explicit information models • Mapping concepts	**Knowledge Representation**
Process aspect • Explicit interaction protocols • Process integration concepts	**Service-Oriented Architecture**
Development aspect • Abstraction • Automation	**Model-Driven Engineering**

Figure 3-37
Relation of problem aspects and proposed solution approaches

In the remainder of thesis, we investigate to which extent and in which way these three solutions approaches can be combined.

Part III.
Solution

Chapter 4

Conceptual Framework for Service Modelling

The objective of this chapter is to define *a conceptual framework for service modelling (COSMO)*. The purpose of the framework is to serve as *common semantic meta-model* that enables the description, integration, verification and simulation of (integrated) service-oriented applications. Using COSMO, one can model the domain of a system, the interactions among its services and their relations, and reason whether these services are interoperable.

This chapter is largely based on joint work with colleagues in the Freeband A-Muse project[26], which was, in turn, based on research on the interaction systems design language (ISDL (Ferreira Pires, 1994)). ISDL has a graphic notation and formal semantics defined in (Quartel, 1997). The main contributions of COSMO with respect to ISDL are the following: first, we extend ISDL with concepts from formal knowledge representation languages. In this way, we enable the formal specification of the information models associated with service-oriented systems. In turn, this enables the automatic reasoning about the interoperability of systems. Second, we structure the concepts of ISDL into different dimensions, enabling the modelling of same services at different levels of abstraction and from different perspectives. In this way, different models of the same service can be used for different purposes such as discovery, integration and implementation. Finally, we evaluate the framework by applying it to two integration cases (presented in Chapter 7 and 8). Note, that ISDL was designed to model the behaviour of distributed software systems whereas, defining COSMO, we specifically focus on service modelling. In this way, COSMO can be used to model not only software but also business services.

Although the author of this thesis has contributed to all aspects of the conceptual framework, his main contributions have been in *extending ISDL*

[26] http://a-muse.freeband.nl/

with formal knowledge representation concepts, contributing to the *structuring of the concepts* and providing an *integration method* that uses COSMO. He also contributed to the validation of the framework by applying it to the integration cases presented in Chapter 7 and 8.

The remainder of this chapter is organised as follows: In Section 4.2, we analyse a number of regularly encountered interpretations of the service concept and use them to derive generic service properties. In Section 4.3, we present concepts to model the identified service properties and structure them into three dimensions – *service aspects*, *abstraction levels* and *perspectives*. Further, each dimension is presented in details. In Section 4.4 we compare COSMO with two closely related conceptual frameworks – one from academia and one from the industry. Finally, in Section 4.5 we present our conclusions.

4.1 Introduction

Service-oriented Architecture (SOA) provides concepts for describing relevant aspects of services and linking services of different systems. It reduces the gap between the problem and the solution domain by providing shared language for business domain experts and software developers to communicate about the problem and its solution. *Model-driven architecture (MDA)*, in turn, enables specification of (service) models at different levels of abstraction (using the SOA concepts). In this way, both domain experts and software developers can address specific concerns at each design step and express their choices using shared set of concepts. In addition, MDA enables different models to be related by transformation activities. In this way, the relation between the models becomes more explicit, which, in turn, enables traceability from requirements to implementation.

The *service* concept has been used implicitly and explicitly in previous paradigms like object- and component-orientation, but not to its full potential. In addition, observing the different existing definitions of the term "*service*", we can conclude that a general definition and understanding of the service concept is still missing.

The service concept should precisely define which system properties have to be modelled, and which not. The selection of properties should be based on the intended use of the service concept for building and integrating (business and IT) systems. For example, one may want to specify a new service by composing existing service specifications at design time, and by using discovery and trading techniques at run-time to find actual realization of these services. In order to support such a scenario, service descriptions should define both the service interactions (to define

service choreographies and orchestration) as well as the purpose and the capabilities of the system (to facilitate discovery and trading).

The framework presented in this chapter defines the basic concepts needed to describe the services of a system. Further, it defines how these basic concepts can be combined to describe more complex service properties.

4.2 The Service Concept

The service concept is widely used in both business and computer science. However, the definition of the concept differs considerably in these areas and even in different "schools of thought" within these areas. This section presents a number of regularly encountered interpretations of the service concept and uses them to derive generic service properties.

Service as value-adding interaction

In economics and business science, services are seen as the *non-material equivalent of goods*. Service provisioning has been defined as *an economic activity that does not result in ownership, and this is what makes it different from providing physical goods*. Service provisioning is claimed to *create benefits by facilitating a change in customers, a change in their physical possessions, or a change in their intangible assets*. The IBM Services Research group defines a service as "*a provider/client interaction that creates and captures value*"[27]. (Quartel, 1997) also uses this interpretation and defines a service as the "*common behaviour of some system and its environment, which is defined in terms of common interactions, the results established in these interactions, and the causal dependencies between them*". (Wieringa, 2003) defines a service as "*an interaction with a triggering event that delivers an identifiable added value to the environment [of the system]*".

Service as capability

Often the service concept is connected to the system or entity providing it. (Baida, 2004) defines a service as "*the capability of a service provider to produce some intangible benefits to its environment*". CBDI Forum applies a similar interpretation to IT services: "*a service is a type of capability described using WSDL*" (Sprott, 2004).

Service as operation

In object-oriented and component-based paradigms, each operation or method defined on an object or component is usually seen as a service of that object or component. A service is a part of the object's behaviour,

[27] http://www.research.ibm.com/ssme/services.shtml

which a client can invoke. In some object-oriented languages such as Java, these operations can be bundled together in an interface specification. Thus, an interface is a collection of service definitions. Confusingly, such a collection of operations is called a service in WSDL. However, the current state of practice in interface definition is that only the signature of each operation is specified. The signature specifies the types of the inputs and outputs of an operation, but not its effect or the relationships between the different operations. Signatures of the addition and multiplication operations on two numbers, for example, are equal, whereas the effects of these operations are quite different. Some extensions that go beyond this simplistic interface definition come from the Semantic Web services community, e.g., OWL-S (Martin, 2004), and outside it, e.g., WS-Agreement (Andrieux, 2005).

Service as application

Web services, but also services in general, are most commonly seen as applications (pieces of software) that can be accessed over the Web. The W3C, for example, uses the following definition (Booth, 2004): "*A Web service is a software system designed to support interoperable machine-to-machine interaction over a network*". However, they also make a distinction between the abstract concept of service and its concrete provider: "*A Web service is an abstract notion that must be implemented by a concrete agent. The agent is the concrete piece of software or hardware that sends and receives messages, while the service is the resource characterised by the abstract set of functionality that is provided*". However, in practice, very often this distinction is not explicitly made.

Service as feature

In the telecommunications domain the term service is usually used to refer to a feature that can be provided on top of the basic telephony service, such as "call forwarding", "call back when busy" and "calling line identification".

Service as observable behaviour

In data communication, a service is traditionally defined as the *observable* (or *external*) behaviour of a system. For example, (Vissers, 1986) defines a service as "*the behaviour of the [service] provider as it can be observed by the [service] users*". In other words, the service of a system is the *set of all possible interactions between the system and its environment and their ordering in time*. Sometimes the external behaviour of a system is divided over more than one interface, where each interface is a part of the system boundary. In this case, a service is the behaviour of the system as it can be observed at a particular interface. If you take this to the extreme and make each interface as small as one operation, you get more or less the same interpretation of "service as operation".

Based on the definitions above, we identify the following general service properties:

Involves interaction

A service *involves one or more interactions* between two (or more systems) - a system that uses the service (called *service user*) and a system that provides the service (called *service provider*). These interactions can be described from two different perspectives: a distributed and an integrated perspective. From a distributed perspective, the participation of the service user (respectively the service provider) is defined abstracting from the participation of the service provider (respectively the service user). This means that the distributed perspective defines the external behaviour that is expected from the service user and the external observable behaviour of the system that provides the service. The integrated perspective defines the joint (integrated) behaviour of the user and provider, abstracting from how the user and provider interact in using and providing the service.

The property that a service involves interaction can be found in all definitions given above. The definitions of *service* as "interaction", "capability," and "observable behaviour" consider this interaction from both a user and a provider perspective. Furthermore, the integrated perspective can also be found in the definition of a service as "interaction". The other definitions mainly focus on the provider perspective.

Provides some value

The execution of a service provides some value to the user and the provider. In case of IT services, this value may only involve "intangible benefits", such as a change in possession of goods and money. For services in general, the value may also involve "tangible things", such as the actual exchange of goods using a transportation service. In the latter example, the value of the service may comprise the intangible change of the ownership of the parcel, as well as the tangible exchange of the goods themselves.

The value of a service is established through the combination of the possible results established in the interactions between the service user and provider. Whether tangible or intangible, in information systems these interaction results are represented using lexical objects (i.e., data types and values).

The property that a service provides some value (or benefit) is made explicit in the definitions of a service as "interaction" and "capability". The other definitions also contain this property, but in implicit form. E.g., they refer to the inputs and outputs of operations, the functionality of some application, a provided feature, or the behaviour (functionality) that can be observed.

Unit of (de)composition
The property that *a service forms a unit of (de)composition* is inherent to the service-oriented paradigm, which fosters the development of services by *composing* other services. Each of the definitions above supports this property. Business processes and supporting applications are composed from or decomposed into services, which define smaller business process or application pieces that may be reused when chosen properly. From a user/provider perspective, such a (de)composition has the form of a set of interacting services, where each service may act as a user, a provider or both. From an integrated perspective, a (de)composition is described in terms of dependencies between services, e.g. temporal or causal relationships.

Based on the identified service properties above we give the following definition of service concept:

Service is a set of related interactions between two or more systems that establishes some identifiable effect which has value for the involved systems.

Usually one of the involved systems plays the role of *service provider* and the others play the role of *service user*. Our service definition closely resembles the ones found in (IBM; Wieringa, 2003; Quartel, 1997). We assume that the established *effects* of the systems' interactions have or create some *identifiable value* for the involved systems.

4.3 Structure of the Framework

We structure the concepts of our framework in three axes as depicted in Figure 4-38. We distinguish four *aspects* (i.e., *communication*, *behaviour*, *information* and *quality*), representing service properties that need to be modelled. This classification corresponds to aspects found in frameworks for enterprise architectures like GRAAL (van Eck, 2004) and ArchiMate (Jonkers, 2004). Further, we distinguish three *abstraction levels* (*effect*, *choreography,* and *orchestration*) at which a service can be modelled. The purpose of the abstraction levels is to enable the specification of *different models of the same service*. In this way, different service models can be used for *different purpose* (e.g., for service discovery, composition or implementation). Finally, we distinguish two *perspectives* (*integrated* and *distributed*) which are used to model the participation of the systems in a service (i.e., service provider and user) (Quartel, 2004).

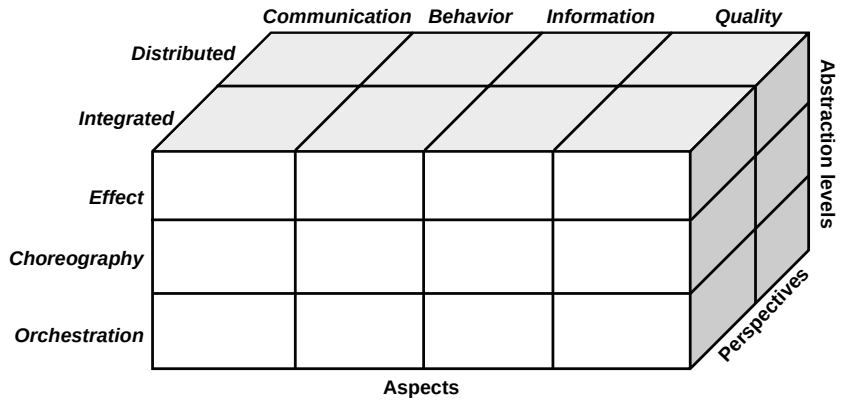


Figure 4-38
The dimensions of COSMO

The purpose of our framework is *to provide concepts for modelling and reasoning about services*. In Chapter 5, we present an integration method that uses the framework to provide complete integration solutions – from business requirements to software implementation. In addition, the method provides for verifying the correctness of the integration solution.

In the following sections, we present each dimension of the framework.

4.3.1 Service Aspects

We distinguish four service aspects, namely *communication*, *behaviour*, *information,* and *quality*.

Communication aspect

The *communication* aspect is concerned with modelling the systems that provide or use services, and their *interconnection structure*. The interconnection structure comprises (amongst others) the *interfaces* at which services are offered.

The *entity* concept models the existence of a logical or physical system. Examples of entities are a university, a Customer Relationship Management (CRM) system, a database management system and a hardware device. We represent entities graphically as a rectangle with cut-off corners (Figure 4-39).

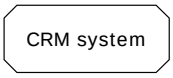
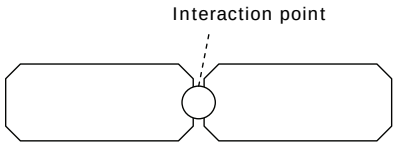


Figure 4-39
Entity

The concept *interaction point* models a shared mechanism that two or more entities may use to interact. An example of an interaction point is a network

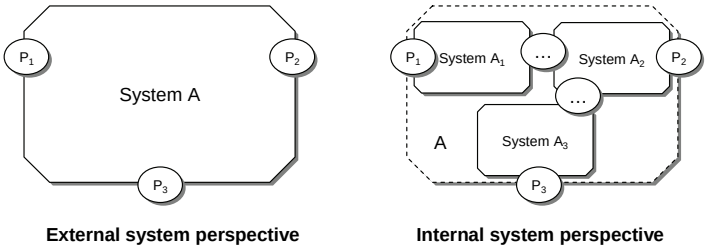
connection between two computer systems. We represent interaction point graphically as a circle that overlaps the systems it connects (cf. Figure 4-40).

Figure 4-40
Interaction point



According to Webster's dictionary a system is "*a regularly interacting or interdependent group of items, components or parts, forming a unified whole*". This definition distinguishes between two system perspectives: an *internal* perspective, i.e., the "*regularly interacting or interdependent group of items, components or parts*", and an *external* one, i.e., the "*unified whole*". Figure 4-41 illustrates the internal and external system perspectives.

Figure 4-41
Internal and external system perspective



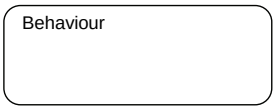
From an external perspective, a system is modelled as a single entity (e.g., *System A*) having one or more interaction points (e.g., P_1, P_2 and P_3), which represent the interaction mechanisms it shares with its environment. The environment of a system is defined as the collection of all systems that share one or more interaction points with that system. From an internal perspective, a system is modelled as a structure of interconnected system parts (e.g., *Systems* A_1, *System* A, and *System* A_3). In this way, the system can be decomposed and its internal structure can be defined. In a similar way, interaction points can be refined into multiple interaction points (Dijkman, 2006).

Behaviour aspect

The *behaviour* aspect is concerned with the activities that are performed by a system as well as the relations among these activities.

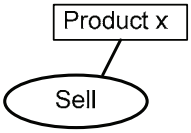
The *behaviour* concept models a group of possibly related activities that a system can perform alone or in cooperation with other systems. Examples of such activities are retrieving customer data from a database, creating a purchase order or transferring money from one bank account to another. We represent behaviour graphically by a rounded rectangle (Figure 4-42).

Figure 4-42
Behaviour



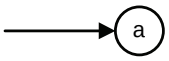
The *action* concept models successful completion of some unit of activity that is performed by a single system. Examples of actions are "selling a product", "sending an e-mail" or "retrieving a database record". An action is *atomic*, i.e., it represents an *indivisible unit of activity*. This means that an action either occurs and has some result (i.e., *effect*) or it does not occur at all and has no intermediate or partial results. In our approach, we model *what* the result of an action is and abstract from *how* this result has been established. An action is graphically represented by an ellipse with the action name placed in the ellipse. The result of an action is defined in a text box, which is connected to the associated action (cf. Figure 4-43). The concept result is explained in detail in the Section "*Information aspect*".

Figure 4-43
Action with a result

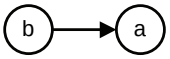


The *causality relation* concept models relationships between different activities. A causality relation defined on action *a* defines the conditions that must be satisfied to enable occurrence of that action. We distinguish three basic causality conditions (cf. Figure 4-44):

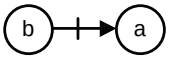
Figure 4-44
Causality conditions



(i) start condition



(ii) enabling condition b



(iii) disabling condition ¬b

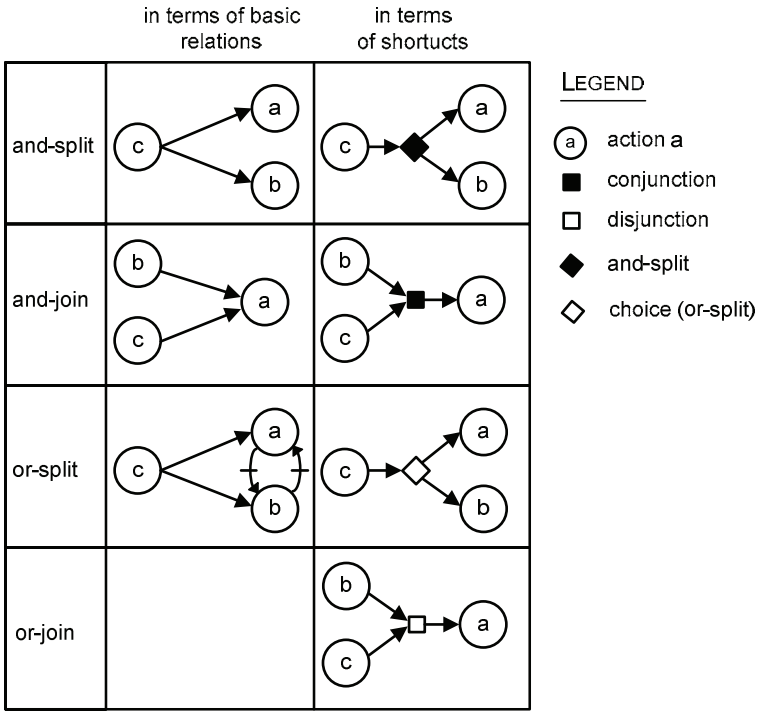
– (i) *start condition* – defines that an activity is enabled and can occur independently of any other activity.

– (ii) *enabling condition* - defines that the occurrence of an activity depends on the occurrence of some other activity. For example, in Figure 4-44(ii), *b* is an enabling condition of action *a*, i.e., action *a* can only occur after action *b* has occurred.

– (iii) *disabling condition* – defines that the occurrence of an activity depends on the non-occurrence of some other activity. For example, in Figure 4-44(iii) ⊣*b* is a disabling condition of action *a*, i.e., action *a* can only occur if action *b* has not occurred before nor simultaneously with action *a*.

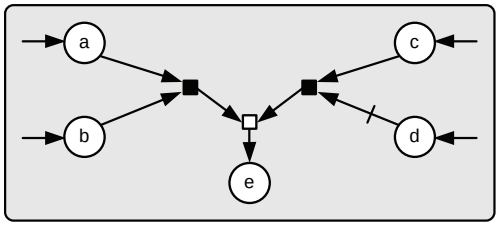
Basic conditions can be combined into more complex ones using operators ■ and □, which define that a *conjunction* and *disjunction* of conditions must be satisfied, respectively. For convenience, we also provide shortcuts to represent *and-split* and (exclusive) *or-split*, ◆ and ◇, respectively (cf. Figure 4-45).

Figure 4-45 Representing the workflow operators



For example, the behaviour in Figure 4-46 defines that actions *a*, *b*, *c* and *d* that are enabled and may occur independently from each other. Action *e* is enabled if and only if both *a* and *b* have occurred or *c* has occurred and *d* has not occurred.

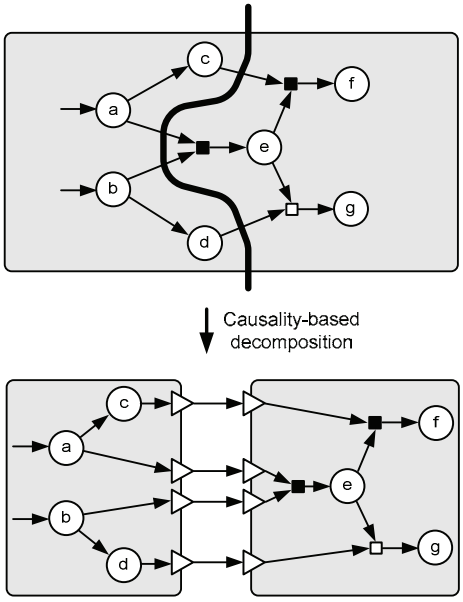
Figure 4-46
Example of complex behaviour



A complex behaviour can be decomposed into smaller and simpler behaviours. This can be done in two ways – a *causality-based decomposition* and a *constraint-based decomposition*.

The *causality-based decomposition* assigns causality conditions of some activity and the activity itself to separate sub-behaviour s (cf. Figure 4-47)

Figure 4-47
Causality-based decomposition



To support causality-based decomposition we introduce two syntactic constructs, namely *behaviour entry* and *behaviour exit*. A behaviour entry represents a causality condition involving causality conditions from one or more other behaviours. A behaviour exit represents a causality condition involving causality conditions only from the behaviour to which the exit

belongs. An exit of one behaviour can be connected to one or more entries of other behaviours.

Constraint-oriented decomposition decomposes an action into interactions and assigns interaction contributions to distinct sub-behaviour s.

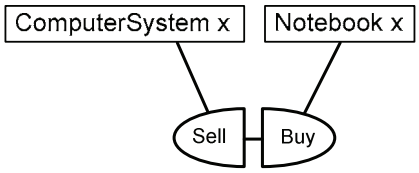
An *interaction* concept models a successful completion of some unit of activity that is performed by multiple systems in cooperation. An interaction can only occur if all participating systems are willing to contribute to the interaction.

An interaction either occurs for all participating systems, or does not occur at all. In case the interaction occurs, all participating systems share same result of the interaction. In case the interaction does not occur, none of the participants can use any intermediate or final results of the interaction. That is, similar to actions, interactions obey the atomicity property.

An *interaction contribution* represents the participation of a system in an interaction, by defining the constraints that this system has on the possible results of the interaction. An interaction contribution is graphically represented as ellipse segment and an interaction as line connecting the flat sides of the involved interaction contributions. Similar to actions, the result of an interaction is defined in text boxes, connected to the respective interaction contributions.

Figure 4-48 depicts an example of an interaction between two systems. The depicted interaction models the activity of selling something, which is performed by the cooperation of two systems (e.g., buyer and seller). Each system contributes to the interaction by the interaction contributions *Sell* and *Buy*. Note that an interaction defines the possible results of the interaction, while abstracting from how these results are established.

Figure 4-48
Interaction



When decomposing an action into an interaction, the conjunction of the causality conditions and result constraints of the interaction contributions must be the same as the causality condition and result constraints of the action. An example of constrained-based decomposition is shown in Figure 4-49.

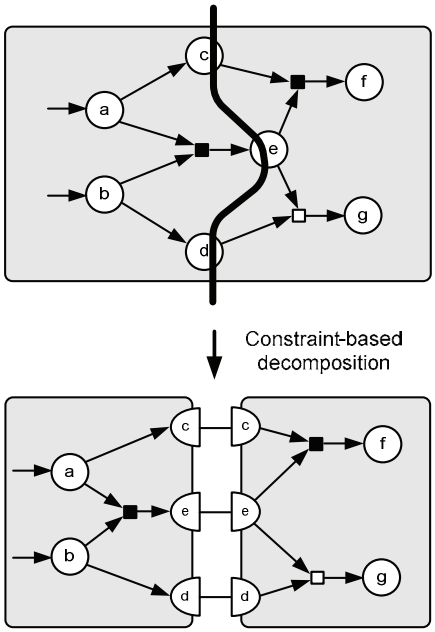
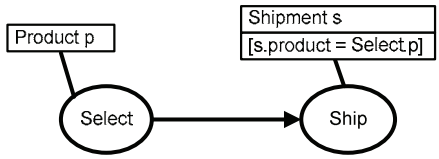


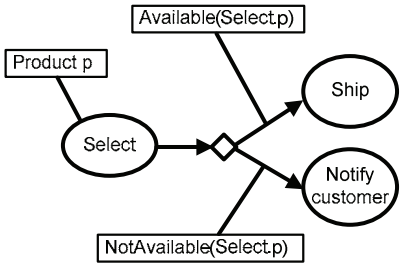
Figure 4-49
Constraint-based decomposition

The result of some activity may depend on the results of other activities. This is modelled by allowing an activity to refer to the results of other activities. For example, in Figure 4-50 the *Shipment* established as a result of the activity *Ship* contains the same product that has been established as a result of the activity *Select*. Analogously, the occurrence of some activity may depend on the result of some other activities. This is modelled by allowing a causal relation for some activity to refer to the results of other activities. For example, in Figure 4-50 the action *Ship* can only occur if this specific product is *Available*. Otherwise, action *Notify customer* occurs.

Figure 4-50 References to results in previous actions



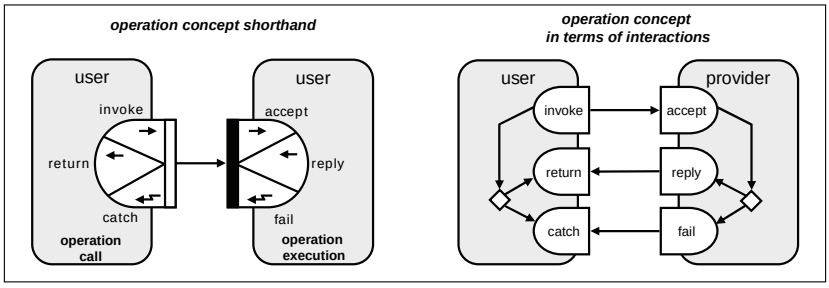
The result of the action *Ship* refers to the result of the action *Select*



The causality relation for the actions *Ship* and *Notify customer* refer to the result of the action *Select*

Some languages, such as WSDL, do not support the basic interaction concept, i.e., they only provide support for modelling (the less expressive) concept of message passing. Therefore, for convenience of system integrators, we provide a new syntactic construct called *operation* (Quartel, 2007) defined as composition of three interactions: *invoke-accept*, *reply-return* and *fail-catch*. Figure 4-51 depicts a shorthand notation for the operation concept.

Figure 4-51 The *operation* concept



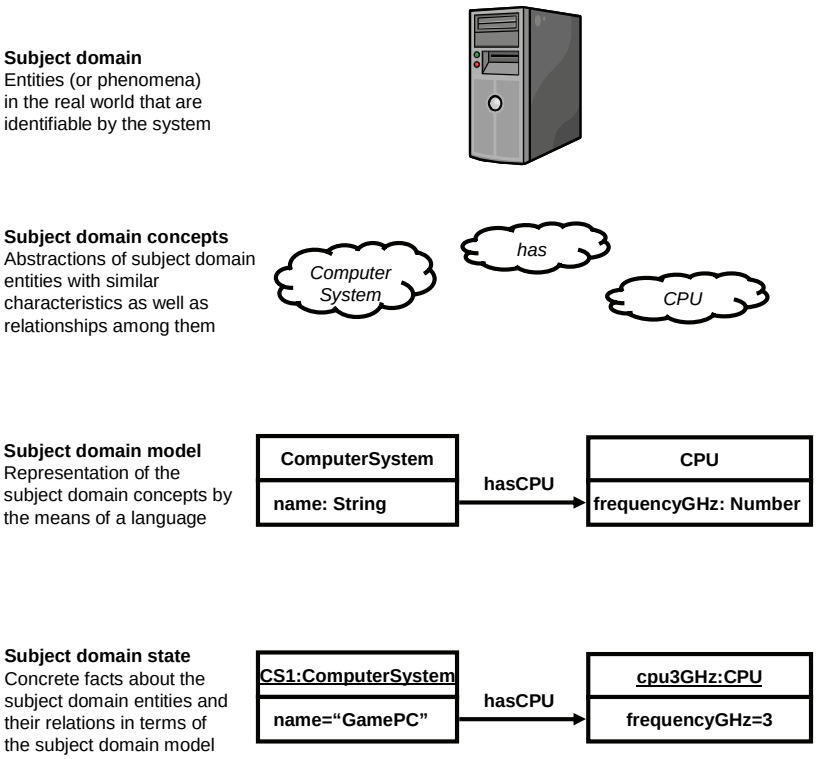
The *reply-return* part and the *fail-catch* part are optional, i.e., either one or both parts can be omitted (e.g., to model a one-way operation).

Information aspect

The information aspect is concerned with modelling the information that is managed and exchanged by a system. Each information system has a *subject domain*. The subject domain comprises the *entities* (or *phenomena*) in the real world that are represented (by the means of lexical entities) in the system.

As discussed in Chapter 2, in order to communicate we abstract real-world entities to *concepts,* and represent these concepts by the means of a *language*. We call such a representation *subject domain model*. All messages that leave a system are constructed in terms of the subject domain model of that system, and all messages that enter the system are interpreted in terms of the subject domain model (cf. Figure 4-52).

Figure 4-52
Subject Domain, Subject Domain Concepts, Subject Domain Model and Subject Domain State



Our conceptual framework does not prescribe a particular language to model the entities (or phenomena) in the subject domain of a system. For illustrative purpose, in this thesis, we use OWL. OWL has been presented in Section 3.3.2. In this section, we recapitulate its main concepts and illustrate how OWL can be used with our conceptual framework.

An *individual* models the existence of a *discrete identifiable part* of the subject domain of the system. Examples are a concrete *person*, *company* or *computer system*. Individuals represent *discrete* parts of the world. This means that we can count them and define the minimum (or maximum) number of individuals (of a certain type) that may exist in the subject domain of a system. Individuals are *identifiable*. This means that counting the individuals in the subject domain does not depend on their state. For example,

withdrawing money from a bank account changes the state of that bank account, but does not change the number of the bank accounts in the bank system.

We use *owl:Class* to represent an abstract type of entities (or phenomena) that share some properties, i.e., an *owl:Class* represents a subject domain concept. When an individual represents an entity that has a type represented by an *owl:Class* we say that the individual is an *instance* of that class. All instances of an *owl:Class* class share some *properties*. For example, persons have *name* and *age*, companies have *employees* and computer systems have *CPU*, *HDD*, *memory* and *display*. Some of the properties have *data values* (e.g., *age* has value positive integer). We model such properties by the means of *owl:DatatypePropery*. Likewise, some properties have *as value an individual* (e.g., an *employee* of a company is a *person*). We model such properties by means of *owl:ObjectPropery*.

To make a property mandatory (i.e., at least one value), to allow only a specific number of values for that property, or to insist that a property must not have any values, we use *cardinality constraints*. OWL provides means for defining three types of cardinality constrains. *owl:maxCardinality N* defines a class of all individuals that have at most *N* (semantically distinct) values (individuals or data values) of a certain property. *owl:minCardinality N* defines a class of all individuals that have at least *N* (semantically distinct) values (individuals or data values) of a certain property. Finally, *owl:cardinality N* defines a class of all individuals that have exactly *N* (semantically distinct) values (individuals or data values) of a certain property.

OWL does not provide a graphical notation. However, tools exist like *Protégé* (Protégé) and *Swoop* (Swoop) that provide means for building OWL ontologies. Nevertheless, in this thesis we illustrate simple information models by means of UML class diagrams as shown in Figure 4-53.

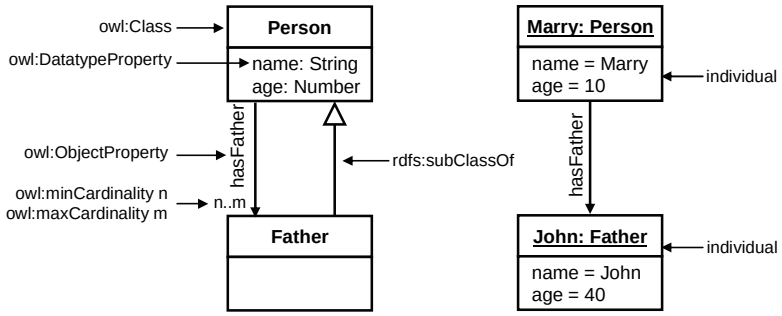


Figure 4-53 Using UML class and object diagrams to represent simple OWL models

The result of an activity is represented in terms of the subject domain models of the systems that participate in this activity.

Figure 4-54 depicts a simplified version of the subject domain model of an online computer shop.

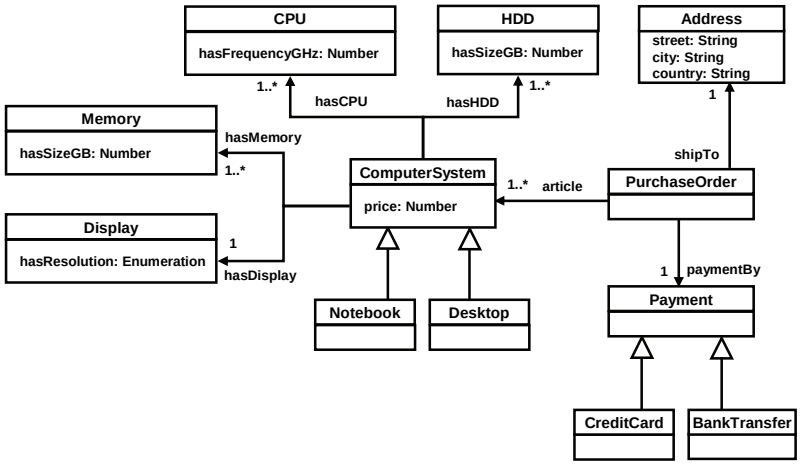


Figure 4-54
The Information model of the online computer shop

The model defines class *PurchaseOrder*. Its instances have properties *article* (of type *ComputerSystem*), *paymentBy* (of type *Payment*) and *shipTo* (of type *Address*). Instances of class *ComputerSystem* have properties *price* (of type *Number*), *hasCPU* (of type *CPU*), *hasHDD* (of type *HDD*), *hasMemory* (of type *Memory*) and *hasDisplay* (of type *Display*). Likewise, instances of class *Address* have properties *street*, *city* and *country* (all of type *String*). Instances of class *CPU* have property *hasFrequency* (of type *Number*), instances of class *HDD* have property *hasSizeGB* (of type *Number*), instances of class *Memory* have property *hasSizeGB* (of type *Number*) and instances of class *Display* have property *hasResolution* (of type *Enumeration*). In addition, our model defines two different types of *Payment*, namely by *BankTransfer* (representing a direct money transfer) and *CreditCard* (representing a payment by a credit card).

Each activity has a *result*. This result is defined using the subject domain model of the system. When an activity occurs, its result is bound to a value (e.g., an instance of a class or a data value).

In addition, a so-called *result constraint* can be defined on the result of an activity. This *constraint* corresponds to a set of predicates that state what properties of the result have to be satisfied by the result value. For example, the interaction contribution *buy* in Figure 4-55 has a result of type *PurchaseOrder* and a result constraint that further specialises possible values of that result. The result constraint defines that the purchase order can only have articles of type *Notebook* with *price at most 1000 euro*. Such a result constraint can also be seen as the *desired effect* that a service user wants to achieve when using a service.

Likewise, the interaction contribution *sell* in Figure 4-55 has a result of type *PurchaseOrder* and a result constraint that further specialises the possible values of that result. The result constraint defines that the purchase order can have articles of type *ComputerSystem* and these articles can be *shipped to*

any city in the Netherlands. Such a result constraint can be seen as the effect that a service provider is capable to achieve.

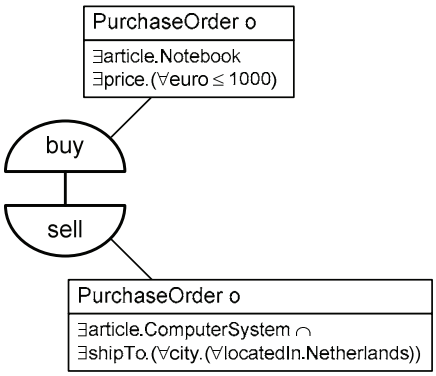


Figure 4-55
The effect model of the online computer shop

The user goal and the provider capabilities are discussed latter in Sections 4.3.2 and 4.3.3.

Quality aspect

The quality aspect is concerned with modelling *non-functional characteristics of services*, which often play an important role in the selection of services. Examples of quality aspects are the "value" that a service has for a user, the "cost" associated with a service and the "response time" of a service. The non-functional properties are outside the scope of this thesis.

The *communication*, *behaviour*, *information* and *quality* aspects represent partially overlapping, i.e., non-orthogonal views on a service. They overlap, because it is generally impossible to specify one aspect without referring to the other ones. For example, to specify certain quality characteristics one must refer to the behaviour, and in order to describe the behaviour, it is necessary to refer to the subject domain model of the system.

4.3.2 Abstraction Levels

As said earlier, we distinguish the three generic abstraction levels at which a service can be modelled, respectively *effect*, *choreography* and *orchestration* level.

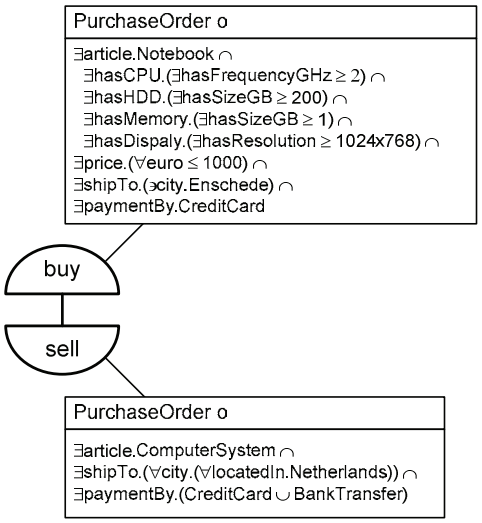
Service effect level

At this abstraction level, a service is modelled as single interaction between two or more systems. The interaction represents an activity in which the involved systems achieve some effect in cooperation. At this abstraction level, we are only interested in what *effect* (i.e., result) can be achieved and not in how it is achieved.

Each system may have different *expectations* on the result of the interaction, and therefore imposes different *constraints* on that result. As said earlier, this is modelled by defining the *interaction as composition of two (or more) interaction contributions*, each representing the participation of the respective system in the interaction.

Figure 4-56 models the example of a computer e-shop service as single interaction between a customer and a retailer. Interaction contributions *buy* and *sell* represent the participation of the customer and retailer in this interaction. The associated text boxes define the constraints they each have on the interaction result. In this case, both the customer and retailer want to establish a purchase order as result of the interaction. The customer wants to order a notebook with a CPU with frequency of at least 2GHz, a hard drive with a size of at least 200GB, a memory with a size of at least 1GB and a display with resolution at least 1024x768, whereas the retailer is willing to sell any computer system. Furthermore, the customer wants to pay with his credit card whereas the retailer accepts two payment options, namely, by credit card or bank transfer. Finally, the customer wants the notebook to be delivered to city Enschede (in the Netherlands) whereas the retailer delivers to any city in the Netherlands.

Figure 4-56
Service effect



The interaction can only occur if the constraints of both the customer and the retailer can be satisfied. In case multiple results may satisfy the constraints (e.g., multiple notebooks may have required properties), only a single result is established. Since the interaction concept abstracts from how to select the result, the latter is assumed to be selected non-deterministically.

Choreography level

At this level, a service is modelled as *multiple related interactions* between two or more systems. The resulting service model defines the *external* behaviour that is requested by the service user and that is offered by the service provider. This model can be used, for example, to specify or analyse interoperability between the service user and provider.

In general, a service cannot be implemented as single interaction and we have to refine the abstract interaction into a structure of multiple smaller more concrete interactions. Figure 4-57 depicts a possible refinement of the example from Figure 4-56 into a number of interactions.

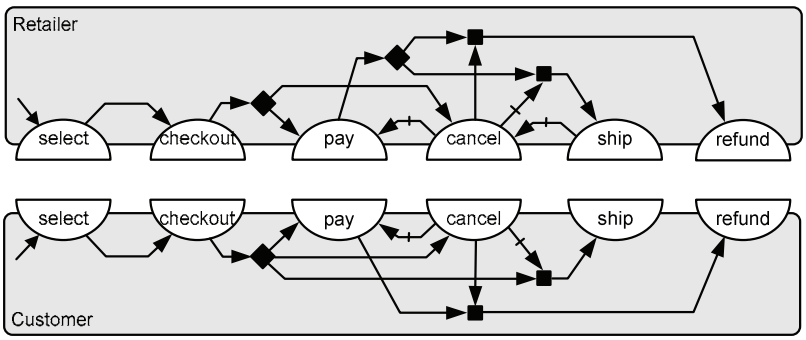


Figure 4-57 Service as a choreography

Interaction *select* represents the mechanism used by the customer to select an article from the product catalogue of the retailer (e.g., a notebook). Interaction *checkout* represents the mechanism used by the user to provide his address and delivery preferences. Interaction *pay* represents the mechanism used by the user to pay for the selected article. Interaction *ship* represents the mechanism used by the retailer to notify the customer that an article has been shipped. Interaction *cancel* represents the mechanism used by the user to cancel an order. Finally, interaction *refund* represents the mechanism used by the retailer to refund the customer in case the payment has been made and the order has been cancelled. For the sake of simplicity, we have omitted results and result constraints of the interaction contributions of the involved systems.

In addition to interaction contributions, at this level of abstraction the customer and retailer may specify the relations among these interaction contributions. In the example from Figure 4-57, the customer has specified the following relations

– *select* does not depend on any other activities and can occur from the beginning of the behaviour

- *checkout* can only occur if *select* has already occurred
- *cancel* can only occur if *checkout* has already occurred
- *pay* can only occur if *checkout* has already occurred and *cancel* has not yet occurred
- ship can only occur if *checkout* has already occurred and *cancel* has not yet occurred
- *refund* can only occur if both *pay* and *cancel* have already occurred

Likewise, the retailer has specified the following relations

- *select* does not depend on any other activities and can occur from the beginning of the behaviour
- *checkout* can only occur if *select* has already occurred
- *cancel* can only occur if *checkout* has already occurred and *ship* has not yet occurred
- *pay* can only occur if *checkout* has already occurred and *cancel* has not yet occurred
- *ship* can only occur if *pay* has already occurred and *cancel* has not yet occurred
- *refund* can only occur if both *pay* and *cancel* have already occurred

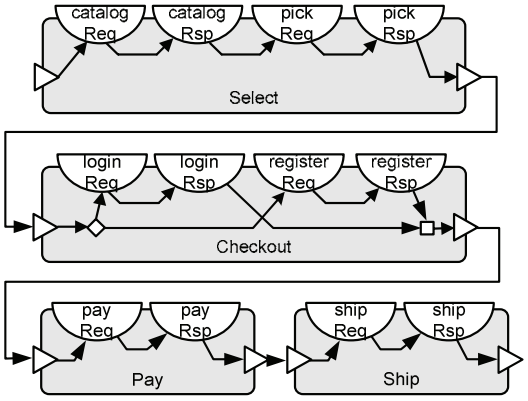
A choreography can be structured into multiple smaller, related choreographies representing groupings of interactions. Typically, such structuring is based on grouping interactions that have strong functional relationships, and separating interactions that have weaker relationships. The aim of this structuring is to increase clarity and comprehensibility of the service definition, to facilitate its mapping onto an implementation, and to separate required from optional functionality. For example, identified groupings may represent suitable units of functionality for searching and selecting existing services or for defining new services that implement part of the required service functionality.

Figure 4-58 depicts an example of a structured choreography. In this example, each interaction from the example in Figure 4-57 is split into two

sub-interactions, a request (*Req*) followed by a response (*Rsp*), such that the result of the response conforms to the result of the original interaction. For example, *payReq* represents a request to perform a payment, and *payRsp* represents the response that informs about the outcome of the payment activity. This type of refinement is needed if one wants to implement the payment interaction using one or more other services. In addition, interaction *select* is further refined by introducing a preparatory interaction *catalog* in which the user can request for a list of articles, followed by an interaction *pick* in which a particular article is selected. Finally, interaction *checkout* is refined to two interactions (*login* and *register*), which allow the customer to login using an existing account or register a new account, respectively.

Sub-choreographies are defined as separate behaviours. To represent causal dependencies between these behaviours, we use *behaviour entries* and *behaviour exits*. For brevity, in the example from Figure 4-58 we only present the retailer's choreography with omitted *cancel* and *refund* interactions.

Figure 4-58 Structured choreography



We use the term interface to provide a perspective on a choreography. Opposed to current practice, we believe that interfaces should also define the relationships between interaction contributions (e.g., operations). Furthermore, a service definition comprising multiple interfaces should also define the relationships between (the interaction contributions from) these interfaces.

Orchestration level

At this level, the service that is offered by some service provider is modelled as *composition* of other services. Typically, the resulting service model defines the service provider as *coordinator* (also called *orchestrator*), which interacts with other service providers and combines the values obtained in these

interactions to offer some added value to the user. This model can be used, for example, to specify or analyse a possible implementation of the offered service.

Besides the refinement of interactions, sometimes it becomes necessary to refine a service into a composition of smaller services in order to obtain an implementation of the service. Figure 4-59 depicts an example of the refinement of the offered e-shop choreography from Figure 4-58 into a number of services: *Catalogue* service that allows one to browse and select articles, *Customer profiles* service that maintains the customers' registrations, *Payment* service that handles payments, a *Shipping* service that is used to package and deliver articles, and *Coordination* service that coordinates the use of aforementioned services to provide the e-shop service.

The *Catalogue*, *Customer profiles*, *Payment* and *Shipping* services are offered services. The *Coordination* service refines the offered e-shop choreography by inserting services that are requested between the procurement interaction contributions. These requested services are used to implement parts of the e-shop choreography. In principle, the *Coordination* service might implement part of the e-shop functionality as well, e.g., order handling. However, in many cases it is considered good practice to provide such functionality by separate services, making the coordination service primarily responsible for coordinating and combining the results of the requested services. This coordination pattern helps to maintain loose coupling between the offered services.

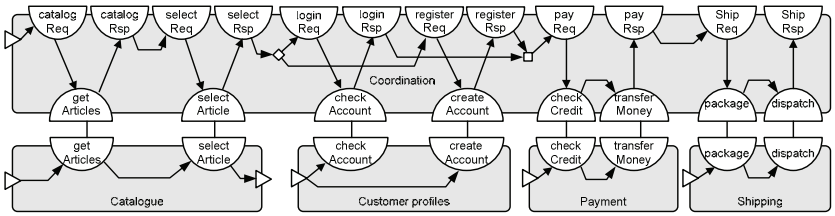


Figure 4-59 Service as an orchestration

As said, the definition of service as a composition of smaller services, including a coordination service, is called an orchestration. In the example above, the orchestration is defined as composition of requested and offered services. Observe that the e-shop interactions have been refined into request and response interactions to model their implementation using other services. By contrast, the interactions of the sub-services do not need this refinement (yet), since the orchestration abstracts from their implementation.

Behaviours in an orchestration can be related using constraint-oriented composition or causality-oriented composition. Similar to constraint- and causality-oriented decomposition, we define *constraint-* and *causality-oriented composition*.

Constraint-oriented composition is used to define two or more interacting behaviours. This composition technique is based on the interaction concept, which decomposes an action into an interaction consisting of two or more interaction contributions. These contributions define the participation of different behaviours in the interaction, which may impose different constraints on the possible interaction results. This allows for an abstract style of service specification and design, i.e., in terms of constraints, thereby abstracting from how these constraints are satisfied by some implementation.

Causality-oriented composition is used to define causal dependencies between behaviours. This composition technique is based on the decomposition of a causality relation, such that an activity and its causality condition can be defined in separate behaviours. For this purpose, entries and exits are used, which represent causality conditions entering and exiting a behaviour, respectively. Like a causality relation associates a causality condition to an activity, an entry point dependency associates a causality condition to an entry point.

During the development process, a service can be modeled successively at the abstraction levels presented above, such that the choreography refines the model of the service as a single interaction, and the orchestration refines the choreography. Furthermore, these abstraction levels may be applied recursively, since the composed services in an orchestration may at first be modeled as a single interaction, and subsequently be refined into choreographies and orchestrations.

The service models at each abstraction level are related through a so-called refinement relation. For example, the orchestration in Figure 4-59 is a refinement of the choreography in Figure 4-57, which is again a refinement of the single interaction in Figure 4-56.

To assess the conformance between service models at the different abstraction levels, we use the method presented in (Quartel, 2007). In this section, we only sketch the main principles of the method. For the detailed presentation of the method, we refer to (Quartel, 2007; Quartel, 1997).

A service model is considered a *refinement* of another service model, if the former model defines additional properties of the service, while preserving the properties defined in the latter model. The opposite of refinement is *abstraction*, which constitutes the process of removing properties.

The assessment method consists of two steps. The first one *determines the abstraction* of the concrete service by abstracting from the service properties that were added in the refinement step. The second step *compares* this abstraction to the original abstract service by checking the equivalence between both abstract models. The refinement is considered correct if both

models are equivalent. Otherwise, the refinement is considered incorrect. The method is illustrated in Figure 4-60.

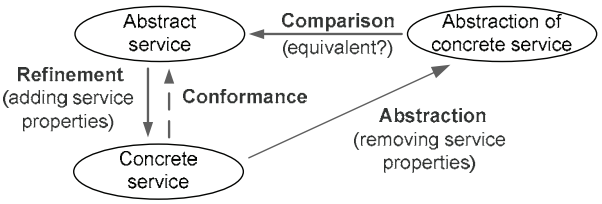


Figure 4-60 Assessment method

To perform the abstraction and comparison step in Figure 4-60 the method provides formal rules, which define how to abstract from service properties and how to compare the abstract service with the abstraction of the concrete service. For a detailed description of the assessment method and the formal definition of the rules we refer to (Quartel, 1997).

4.3.3 Service Perspectives

We use the term ”system” in its general meaning, representing, for example, people, organizations, software applications or hardware devices. A system may be involved in multiple services, and may even act as user of one service and as provider for another. Therefore, we cannot say that a system is either service provider or service user. Furthermore, the specific system that provides some service may not be known at design time or even at discovery time. For these reasons, we currently do not model the involved systems explicitly. Instead, we model the role of the system in a service, where we distinguish two roles: the user role and the provider role.

The user and provider roles define a service from a *distributed* perspective. The *user role* defines the participation of the user in the service, representing the expectations the user has on the effect of the service. This partial definition of the service is also called the *requested service*. The *provider role* defines the participation of the provider, representing the expectations it has on the user. This partial definition of the service is also called the *offered service*. Finally, the *integrated* service perspective defines the joint (integrated) behavior of the user and provider, abstracting from the particular choice on how the user and provider participate and cooperate in performing the interactions. The action concept is used to represent a joint activity (integrated interaction) by abstracting from the distinction between the user and provider roles.

Figure 4-61 summarises the different abstraction levels and perspectives.

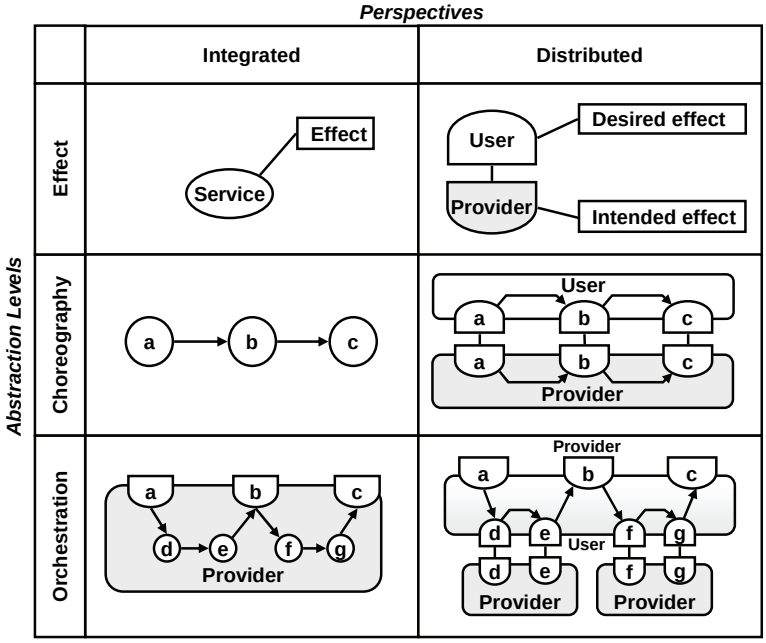


Figure 4-61
Service levels and perspectives

4.4 Comparison

There are some ongoing related efforts in creating a conceptual framework for service modeling. The most prominent examples are the W3C's Web Services Architecture (WSA[28]), Colombo (Curbera, 2005), OWL-S (Martin, 2004), *Web Services Modeling Ontology* (WSMO (Bruijn, 2005)), OASIS's SOA-RM[29] (Estefan, 2008) and SoaML[30].

In this section, we compare COSMO with two of these frameworks – one from academia (WSMO) and one from industry (SOA-RM). For that purpose, we use the *feature comparison method* (Siau and Rossi, 1998), i.e. we define a number of *evaluation criteria* and analyse each conceptual framework using these criteria. Note, that this comparison technique is subjective. First, defining evaluation criteria is a very subjective task. Second, interpreting the descriptions of the compared framework is also a subjective task. The strength of this comparison approach is that it is relatively easy to perform.

[28] http://www.w3.org/TR/ws-arch/
[29] http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf
[30] http://www.omg.org/docs/ad/08-08-04.pdf

4.4.1 Evaluation Criteria

To evaluate a conceptual framework for service modeling, we need to evaluate what aspects of services can be expressed using the framework, and how easily this can be done. Besides, we need to evaluate how generic the defined concepts are. For that purpose, we define an evaluation framework that consists of three dimensions as shown in Figure 4-62. Our framework is derived from the framework for evaluating business process modeling languages and tools presented in (Janssen, 1997).

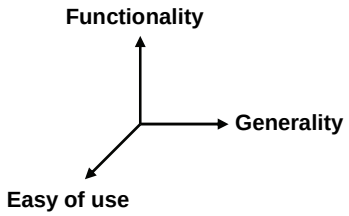


Figure 4-62 Dimensions of the comparison framework

The functionality dimension includes all (technical) capabilities of a service modeling framework. When evaluating the functionality we consider the following criteria:

- *Expressiveness*: Do the concepts enable to model all relevant service aspects (e.g., structure, information, behavior, and value)?
- *Structuring*: Does the framework offer structuring techniques (e.g. composition and decomposition, abstraction and refinement, and modularity and encapsulation)?
- *Formality*: Do the concepts have a formal foundation?
- *Analyzability*: Which types of analysis can be performed on a model?
- *Relevance*: How appropriate are the modeling concepts in the context of service modeling?

The "ease of use" dimension includes the following criteria:

- *Accessibility*: Are the concepts comprehensible?
- *Usability*: How easily can a service be modeled? Does the framework offer pre-defined constructs and high-level concepts?

– *Adaptability*: How easily can the concepts be adapted to individual needs?

– *Openness*: Can a language or tool based on these concepts be used in combination with other languages and tools?

Finally, the "generality" dimension includes the following criteria:

– *Domain independence*: Are the concepts independent from any specific business domain or application?

– *Language independence*: Can the concepts be related to existing (service) modeling languages?

In the following sub-sections, we present the two most closely related service modeling frameworks and use the presented evaluation criteria to compare them with COSMO.

4.4.2 The Web Services Modeling Ontology

The *Web Service Modeling Ontology (WSMO (*Bruijn, 2005*))* is a formal ontology for describing several aspects of Semantic Web Services. It consists of four main components – *Ontologies*, *Goals*, *Web Services* and *Mediators*. *Ontologies* provide terminology and formal semantics of information that is used by the other components. A *goal* is a specification of the objectives of a service user. A *Web service* is a specification of the functionality of the service provider. *Mediators* are used as connectors between ontologies, goals and web services.

Both goals and web services are described in terms of used *ontologies*, *interfaces*, *desired capabilities,* and *non-functional properties*. A *capability* specifies what a service does. It is defined in terms of *preconditions* (state of the system before the service execution), *assumptions* (state of the environment before the service execution), *postconditions* (state of the system after the service execution) and *effects* (state of the environment after the service execution). An *interface* specifies how the functionality of the service can be used. It defines the *choreography* and *orchestration* of a service. The *choreography* describes the interactions between the service requestor and the service provider. The *orchestration* describes how the service makes use of other services to achieve its capability.

4.4.3 Reference Model for Service Oriented Architecture

The OASIS SOA-RM (Estefan, 2008) defines service as "*a mechanism to enable access to one or more capabilities, where the access is provided using a*

prescribed interface and is exercised consistent with constraints and policies as specified by the service description".

SOA-RM is partitioned into three views:

- *Business via Services* - focuses on how people conduct their business using SOA. This view includes *Stakeholders and Participants Model*, *Needs and Capabilities Model*, the *Resources Model*, and the *Social Structure Model*.

- *Realizing Service Oriented Architecture* - focuses on the infrastructural elements needed to support the construction of SOA-based systems. This view includes *Service Description Model*, *Service Visibility Model*, *Interacting with Services Model*, *Realization of Policies Model*, and *Policies and Contracts Model*.

- *Owning Service Oriented Architectures* - focuses on aspects concerning owning, managing and controlling a SOA.

Both COSMO and WSMO do not provide explicit concepts to define models at *Business via Services* and *Owning Service Oriented Architecture* views. For that reason, we skip the detailed presentation of these two views and focus on the *Realizing Service Oriented Architecture* view.

A service *description* contains information about service *reachability*, service *interface*, service *functionality* and all related *policies*, *contracts* and *metrics*. Service *reachability* describes the *endpoints* of a service and the *protocol* to be used for message exchange using a specific endpoint. Service *interface* defines the means for interacting with a service. It describes the *information* and *behavioral model* of the service. The *information model* defines the *structure* and the *semantics* of the messages that can be exchanged with the service. The *behavioral model* defines the *actions* that can be performed by the service and the *correct temporal order* of their execution. Service *functionality* describes what can be expected when interacting with a service. It is an unambiguous expression of service *functions*, *technical assumptions* and the *real world effects* of invoking the function. *Policies* prescribe the *conditions* and *constraints* for interacting with a service. The differences between *technical assumptions* and *policies* are that whereas *technical assumptions* are statements of physical facts, *policies* are *subjective* assertions made by the service provider or from higher authorities. *Contracts* are agreements among the service providers and service users. A *contract* may reconcile inconsistent *policies* asserted by service participants or may specify some further details of the interaction. For example, *service level agreements* (*SLAs*) is one of the most commonly used category of *contracts*. Policies and contracts are tracked in *compliance records*. *Metrics* provide *operational values* for these *compliance records*. They identify

performance quantities that characterise the speed and quality of realizing the real world effects by the service or non-performance metrics, such as whether a license is in place to use the service.

4.4.4 Comparison

In this sub-section, we compare COSMO, WSMO and SOA-RM using the evaluation criteria defined in Section 4.4.1.

With respect to *expressiveness*, all frameworks enable the modeling of relevant service aspects as *structure*, *information* and *behavior*. COSMO uses *causality* instead of the *data flow* to model the relationship between activities performed by some service. In this way, it provides greater expressiveness allowing one to model concurrency, independence, and disjunction which is difficult or impossible using purely flow-based concepts. Besides, COSMO does not prescribe a particular language for expressing an information model of a service. In this way, COSMO can be used with all popular knowledge representation languages such as ERD, UML or OWL, providing a desired degree of expressiveness. However, neither COSMO nor WSMO provide explicit concepts to model how people conduct their business nor to express governance, security and service management models.

With respect to *structuring*, COSMO provides better techniques comparing to WSMO and SOA-RM. First, COSMO allows a behavior model to be decomposed into sub-behaviors in two different ways: using a *causality-based* and *constraint-based decomposition*. Second, providing the *behavior* concept, COSMO allows for *modularity* and *encapsulation*. Finally, COSMO enables the same service to be modeled at *different levels of abstraction* and from *different perspectives*. Service models at different abstraction levels can be used for *different purposes* such as service discovery and composition. The *integrated service perspective* is useful *to reason about the service as whole*, e.g., by abstracting from the participation of each system in the service.

With respect to *formality*, both WSMO and COSMO provide concepts having a formal foundation. WSMO uses *F-Logic* to provide formal semantics for its information modeling concepts and *Abstract State Machines* (*ASM*) for the behavior modeling concepts. COSMO has formally defined behavior modeling concept (defined in (Quartel, 1997)). It does not prescribe any particular knowledge representation language, i.e., the information models can be specified using formal knowledge representation techniques such as OWL or F-Logic.

With respect to *analyzability*, both COSMO and WSMO enable formal reasoning about service models, i.e., they support reasoning task such as consistency checking, detection of deadlocks and reachability analysis. In

Chapter 5, we define a method that uses the analyzability of COSMO models to check formally whether a number of systems are interoperable.

With respect to *relevance*, all frameworks provide appropriate modeling concepts. SOA-RM is stronger in modeling the social aspects (such as service stakeholders and participants) as well as service management and governance aspects.

With respect to *accessibility*, all frameworks provide concepts that are comprehensible. COSMO has fewer number of concepts than the other frameworks, but at the same time provides comparable expressive power. WSMO provides an explicit concept for *mediator*, for example. In fact, this is a merely *syntactic construct.* In COSMO, an *interaction* can be used to represent a WSMO *goal-to-goal mediator*. *Goal-to-web-service* and *web service-to-web* service *mediators* correspond to *refinement steps* in COSMO, where one interaction is decomposed into a choreography. The advantage of COSMO is that one can reason about the semantics of mediators, e.g., the matching of goals in terms of interaction constraints, and the relationship between goals and web-services in terms of conformance relations.

With respect to *usability*, COSMO has a graphical notation (adopted from ISDL) which allows a service model to be expressed in a graphical way. This is particularly useful when the structure of a service model needs to be made explicit. In addition, COSMO has a formal metamodel. This allows existing tools, such as *Graphical Modeling Framework (GMF)*[31] or *openArchitectureWare (OWA)*[32], to be used to develop graphical or textual domain specific languages (DSLs), which in turn, significantly increases the usability and productivity of the service models. Finally, COSMO provides predefined, higher-level concepts such as service *operation* that additionally increase the usability.

With respect to *adaptability*, COSMO is stronger than WSMO and SOA-RM. COSMO concepts can be easily adapted to individual needs. For example, in the A-Muse project (A-Muse, 2008) COSMO concepts have been adapted to create a DSL for modeling context-aware, mobile services. For that purpose, COSMO concepts have been adapted to support modeling of *context events*, *user input* and *output*.

With respect to *openness*, both COSMO and SOA-RM allow their concepts to be mapped to existing service modeling languages such as WSDL and WS-BPEL. In Chapter 7 and 8, we show how COSMO can be used in combination with WS-BPEL and Java. WSMO also claims openness, however, so far it has been used only in the context of Web Services.

Finally, all frameworks provide comparable *domain* and *language independence*. Note, that WSMO prescribes F-Logic to express service

[31] http://www.eclipse.org/modeling/gmf/
[32] http://www.openarchitectureware.org/

models. In some cases, the formalization of service models may significantly increase their complexity and unnecessarily decrease the usability. In addition, WSMO assumes Web Services as implementation technology, whereas COSMO and SOA-RM models can be implemented using any programming language.

Figure 4-63 summarises the comparison of COSMO, WSMO and SOA-RM.

Figure 4-63 Summary of the comparison

	WSMO	*SOA-RM*	***COSMO***
Functionality			
•Expressiveness	+	++	+
•Structuring	-	+	++
•Formality	++	-	++
•Analyzability	++	-	++
•Relevance	++	++	++
Ease of use			
•Accessibility	+	-	++
•Usability	-	+	++
•Adaptability	-	-	++
•Openness	+	++	++
Generality			
•Domain independence	+	++	++
•Language independence	+	++	++

- bad, + good, ++ very good

We can conclude from the table that all frameworks provide *comparable* functionality, ease of use and generality. COSMO is *stronger structuring, usability and adaptability*.

4.5 Conclusions

Although service-orientation is widely recognised as a promising approach to deal with the complexity of IT systems, so far, its central concept "service" has not been used to its full potential due to the lack of a comprehensive conceptual framework.

Based on an analysis of commonly found interpretations of the service concept, we identified *general service properties*. Using a simple example, we introduced and illustrated *basic concepts* that support the identified properties and underlie the service concept. Moreover, these basic concepts helped us to explain, relate and in fact formalise important notions, such as service *effect*, *choreography* and *orchestration*.

The key properties of our framework are:

- the framework is constructed from a *small number of basic concepts*, which are *based on practice*, but at the same time provide a powerful conceptual basis for service modeling;

- the framework is *language-independent*, but at the same time the basic concepts of the framework can be related to many of the popular languages used in the context of service design, analysis and implementation;

- the framework is *domain-independent*, i.e., no assumption is made with respect to the type of systems for which services should be modeled. We expect that our framework will have a wide spectrum of application, for example, it can be used to model services at a business, application and component level, thus beyond the usual domain of web services;

- the framework is particularly strong in the modeling of services at *different abstraction levels*. We identified three generic abstraction levels, namely, *service effect*, *choreography* and *orchestration*.

By defining the framework, we answered *Research question Q3*: "How to model the semantics of a service? What aspects of services should be modelled and how? At which abstraction levels? How can we use these concepts to reason about a service?". In the following chapter, we propose a method for service integration that uses the conceptual framework presented in this chapter. The framework and the integration method are validated in Part IV of this thesis.

Chapter 5

Model-Driven Service Integration

In this chapter, we present *a method for the semantic integration of service-oriented applications*. The chapter is organised as follows: first, we identify *necessary conditions* for semantic and pragmatic interoperability of service-oriented applications. Next, we propose *a model-driven integration method* that uses *semantically enriched service descriptions* to deliver end-to-end integration solutions from business requirements to software implementation. Finally, we present a method to *verify formally* whether the proposed integration solution meets the identified conditions for interoperability.

5.1 Necessary Conditions for Interoperability

In Chapter 3, we have identified three levels of interoperability, namely *syntactic*, *semantic* and *pragmatic* interoperability. We have further discussed what interoperability problems can occur at each of these levels. In this section, we continue this discussion and define *necessary conditions for interoperability*. The identified necessary conditions are used later to check whether a number of system to be integrated are interoperate.

5.1.1 Syntactic Interoperability

Syntactic interoperability is concerned with ensuring that systems, involved in some interaction use the same *vocabulary* and *grammar* to construct and parse the messages they exchange.

Web Service standards address syntactic interoperability by providing XML-based standards such as SOAP, WSDL and WS-BPEL. XML is a platform-independent mark-up language capable of describing both data and data structure. This way, different systems can parse each other's messages, check whether these messages are well-formed, and validate whether the messages adhere to a specific syntactic schema. In this thesis

we adopt XML-based standards to deal with the syntactic interoperability problem and focus only on semantic and pragmatic interoperability.

5.1.2 Semantic Interoperability

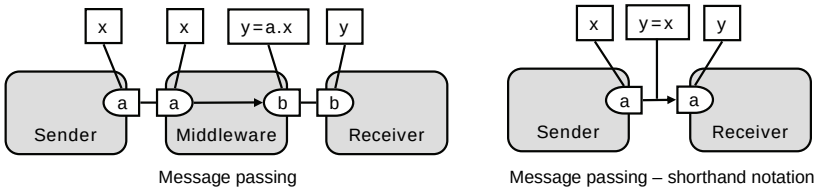
Semantic interoperability is concerned with ensuring that a symbol has the same *meaning*, (i.e., refers to the same thing in the real world) for all systems that use this symbol in their languages.

Systems exchange messages that consist of property values of entities (or phenomena) in their shared subject domain. *Semantic interoperability problems* arise when different systems use *different symbols* to refer to *same things* in the real world or use the *same symbol* to refer to *different things* in the real world. Such systems can interoperate if the data in the exchanged messages is translated in terms of the respective subject domain models. The translation is captured by *source-to-target mappings* that specify *what* data from the sent message should appear in the received message and *how*.

Let *S* be the information model of a system that sends a message. Let *T* be the information model of a system that receives a message. Let *M* be a set of source-to-target mappings defined as predicates on the elements of *S* and *T*. Then: if *x* is a message sent by the first system and *y* is the respective message received by the second system, the following condition has to be met:

> ***Necessary condition 1****: A necessary condition for semantic interoperability of two systems is that the sent message x is a valid instance of S, the received message y is a valid instance of T and the sent and the received messages together satisfy the predicates defined by the source-to-target mapping M.*

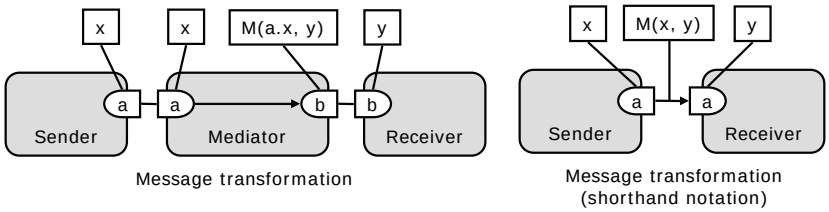
Figure 5-64
Message passing



The systems will be interoperable per-se if no message translation is needed to produce the received message from the sent message, i.e., the middleware is only responsible for transporting the message (cf. Figure 5-64). However, in many cases the sent message has to be translated such that the systems can interoperate. In some cases, a message even has to be split into multiple messages or be combined with other messages to form a new message understandable by the receiving system. The translation functions are performed by an intelligent middleware, called *Mediator* that is not only

responsible for transporting messages but also for translating them, i.e., source-to-target mappings can be satisfied.

Figure 5-65 Message transformation



For example, suppose two systems use different unit systems to construct and interpret the messages they exchange, e.g., the first system measures and reports the speed of a vehicle in miles per hour. The second system takes the speed measurement assuming kilometers per hour and will issue a warning if the speed exceeds the maximum allowed speed of the current road segment. Let the message sent by the first system be

$$source{:}CurrentSpeed = 55$$

and the maximum allowed speed for the current road be 80 km/h. Without any translation of the message, the second system will not issue a warning because the value 55 is less than 80. To compensate this problem the mediator must translate the message in terms of the subject domain model of the second system, e.g. from miles per hour to kilometres per hour. The translation is captured in the following source-to-target mapping

$$\forall x\ source{:}CurrentSpeed(x) \wedge \forall y\ target{:}CurrentSpeed(y) \rightarrow y = x * 1.609344$$

This way, when the first system reports a speed measurement of 55 miles per hour the second system will receive the following message

$$target{:}CurrentSpeed = 88.51392$$

Since the value 88.51392 is bigger than 80 (the maximum allowed speed for the current road segment) the second system will issue a warning.

In Section 5.2, we present a method for building mediators.

5.1.3 Pragmatic Interoperability

Pragmatic interoperability is concerned with ensuring that message sender and receiver share the same expectation about the *effect* of the exchanged messages.

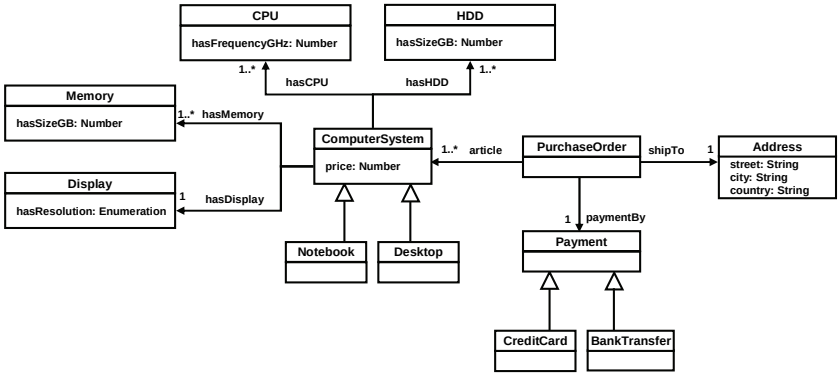
When a system receives a message it changes its state, sends a message back to the environment, or both (Wieringa, 2003). In most cases, messages sent to the system change or request the system state, and messages sent from the system change or request the state of the environment. That is, the messages are always sent with some intention for achieving some desired effect. In most cases, the effect is realised not only by a single message, but by a number of messages sent in some order. Pragmatic interoperability problems arise when the intended effect differs from the actual one.

Our conceptual framework allows system designers to specify the possible results of a system interaction by defining constraints on the result of the interaction.

> ***Necessary condition 2****: A necessary condition for pragmatic interoperability of an interaction is that at least one result that satisfies the constraints of all contributing systems can be established.*

To illustrate this condition we use the example of the online computer shop presented in Chapter 4. The information model of the system is presented in Figure 5-66.

Figure 5-66
The information model of the online computer shop system



The interaction contributions *buy* and *sell* (cf. Figure 5-67) represent the participation of the customer and retailer in this interaction. The interaction contribution *buy* defines a class of acceptable purchase orders for the consumer, and the interaction contribution *sell* defines a class of acceptable purchase orders for the retailer. In this example, the systems can interoperate because they can establish results that are instances of both classes at the same time.

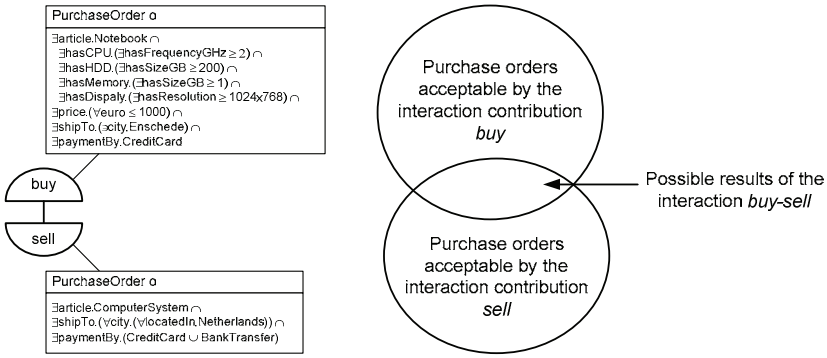


Figure 5-67 Example of necessary condition 2: possible results of the interaction *buy-sell*

In the example above the customer's and retailer's systems can interoperate because the interaction *buy-sell* can establish a result (cf. Figure 5-68).

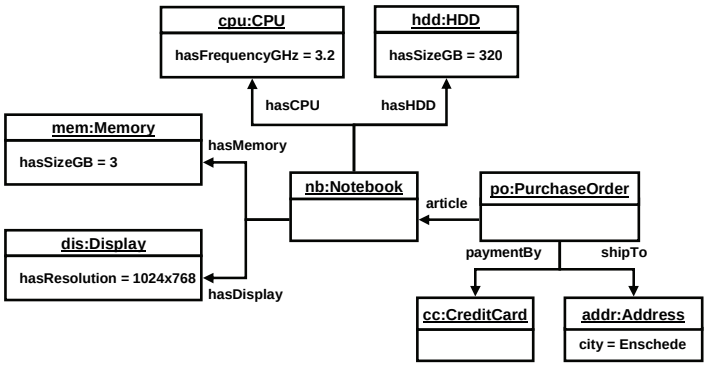


Figure 5-68 Example of a possible result of the interaction *buy-sell*

Very often, a service is not a single interaction but a set of related interactions between the system and its environment.

> ***Necessary condition 3****: A necessary condition for pragmatic interoperability of a service is the existence of at least one execution scenario that can establish all required results.*

Formulating this condition, we use the same online computer shop example (cf. Figure 5-69)

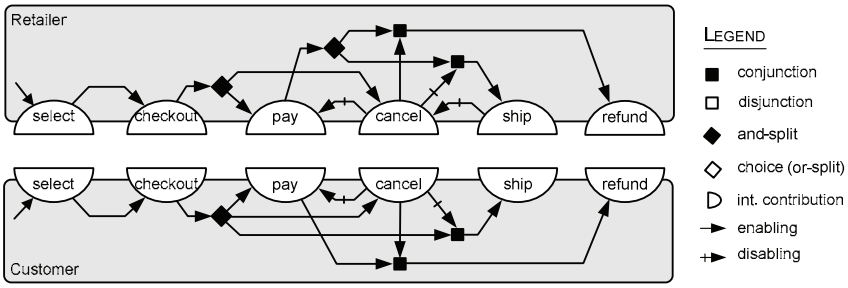


Figure 5-69 Example of Necessary Condition 3

To recapitulate, in this the example, the *Customer* has specified the following relations among its interaction contributions:

- *select* does not depend on any other activities and can occur from the beginning of the behavior
- *checkout* can only occur if *select* has already occurred
- *cancel* can occur only if *checkout* has already occurred
- *pay* can only occur if *checkout* has already occurred and *cancel* has not yet occurred
- *ship* can occur only if *checkout* has already occurred and *cancel* has not yet occurred
- *refund* can occur only if both *pay* and *cancel* have already occurred

Likewise, the *Retailer* has specified the following relations among its interaction contribution:

- *select* does not depend on any other activities and can occur from the beginning of the behavior
- *checkout* can only occur if *select* has already occurred
- *cancel* can occur only if *checkout* has already occurred and *ship* has not yet occurred
- *pay* can only occur if *checkout* has already occurred and *cancel* has not yet occurred
- *ship* can occur only if *pay* has already occurred and *cancel* has not yet occurred

– *refund* can occur only if both *pay* and *cancel* have already occurred

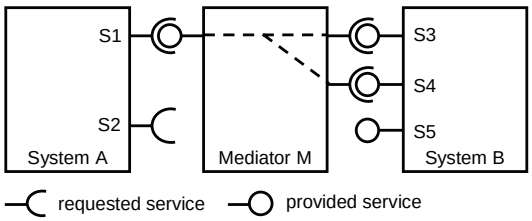
In this example, the retailer and the customer can interoperate because there is an execution trace which is acceptable for both parties, for example, *select-checkout-pay-ship*.

In Section 5.3, we present a method for verification of service interoperability.

5.2 Integration Method

We approach the design of a mediator as a *composition* problem: each service that is requested by some system has to be composed from one or more services that are provided by the other systems and, possibly, by the same system. Figure 5-70 illustrates this for the case of two systems. *Mediator M* offers a mediation service that matches requested service *S1* of *System A* by composing services *S3* and *S4* offered by *System B*. The *Mediator M* should provide such a mediation service for each service that is requested by *Systems A* and *B*.

Figure 5-70
Mediation as service composition



To support the design, implementation and verification of mediators we have developed an *integration method*. Our method uses the COSMO framework presented in Chapter 4 to model and reason about services. It further defines a number of steps to build end-to-end integration solutions and to verify their correctness. In this section we present the steps of the integration method (cf. Figure 5-71). For the sake of readability, we consider only two systems, but the same steps apply to the case of multiple systems.

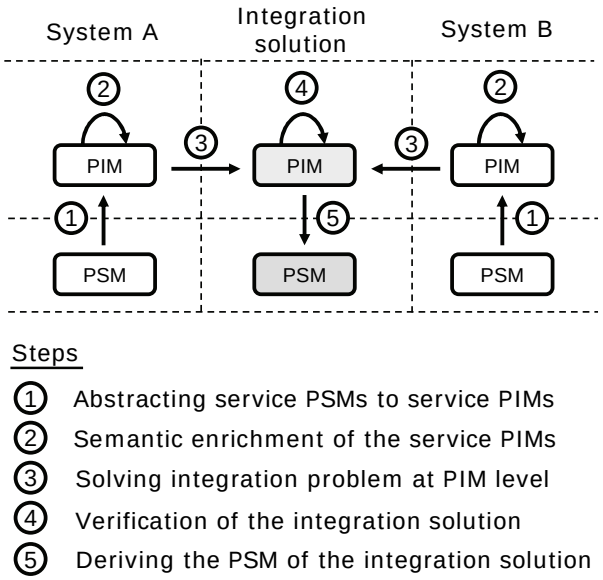


Figure 5-71 Steps of the integration method

In Step 1 of our method, we derive the platform-independent models of the services being integrated by abstracting all technical details from their descriptions. Next, in Step 2 we increase the coverage and precision of these models by adding additional semantic information that cannot be derived from the service descriptions. In Step 3, we solve the integration problem at a technology-independent level which enables the more active participation of the domain experts. In addition, the semantically enriched service models allow some integration tasks to be fully or partially automated. Besides, the abstract nature of the integration solution allows its reuse for different implementation technologies. Next, in Step 4 we formally verify the correctness of the integration solution using automatic reasoning. Finally, in Step 5 the platform-independent service model of the integration solution is transformed to a platform-specific solution by adding technical details by the IT experts. The steps of the integration method are presented in detail in following.

5.2.1 Step 1. Abstracting Platform-specific Service Models to Platform-independent Service Models

In Step 1 (cf. Figure 5-72) of our method, we abstract the service descriptions of the systems to be integrated from implementation-specific information.

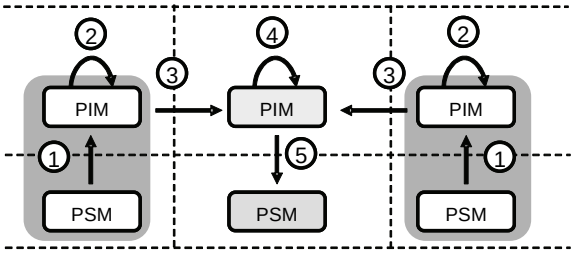


Figure 5-72
Step 1. Abstracting service PSMs to service PIMs

In general, the services of *System A* and *B* are described at implementation (technology) level (e.g., using WSDL). The implementation details may unnecessarily complicate the design of the mediator, and therefore hinder the participation of business domain experts who do not (want to) know how integration requirements are implemented by the means of some specific technology. In terms of MDA, this means that we transform the platform-specific service models (service PSMs) of the systems A and B to their respective platform-independent service models (service PIMs). In Chapter 6 we make this step more concrete by presenting a transformation from service descriptions specified in WSDL to COSMO.

5.2.2 Step 2. Semantic Enrichment of the Platform-independent Service Models

In step 2 (cf. Figure 5-73), the platform-independent service models may be semantically enriched by adding information that cannot be derived (automatically) from the platform-specific service models.

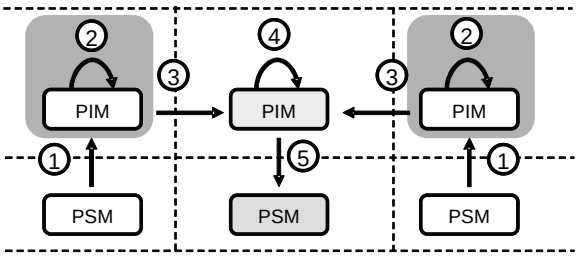


Figure 5-73
Step 2. Semantic enrichment of the service PIMs

The purpose of the semantic enrichment step is to make service models more precise and increase their coverage, which in turn is a necessary condition to reason about and (semi-) automatically derive the mediation service.

The semantic enrichment step involves two activities – (i) semantic enrichment of the service *information models* and (ii) semantic enrichment of the *behavior models* of the systems to be integrated.

Semantic enrichment of the service information models

A service description (e.g., in case of WSDL) usually defines the service operations and the data types of the input and output messages of these operations. In most of the cases, such a service description defines only the *syntax* of the messages, but not their *semantics*. In addition, a service description usually does not capture the implicit assumptions made about the subject domain of a system. To allow for the correct definition of the integration solution and the verification of its correctness, the information models of the systems to be integrated have to be semantically enriched using information from alternative sources. Such sources can be service descriptions in natural language, interviews with business domain experts, or even inspections of the implementations of the services and the databases they use.

The semantic enrichment of the service information models is done by defining new classes, properties and relations. Furthermore, the meaning of some classes and their properties can be defined by mapping them onto domain-specific standards such as *Universal Data Element Framework* (UDEF), *Global Individual Asset Identifier* (GIAI), *Global Location Number* (GLN), *Global Trade Item Number* (GTIN) and *Serialised Shipping Container Code* (SSCC).

The benefit of the semantic enrichment can be fully exploited when using formal knowledge representation technologies that allow one to formally model and reason about the semantics of classes and their properties.

Semantic enrichment of the behavior models

In many cases, a service description does not define the interaction protocol of the service provider, i.e., the ordering of interactions between the system and its environment. Therefore, again, to derive the complete behavior of a system, the system integrator has to use alternative sources of information.

The semantic enrichment of the behavior models is done by defining the relations among service operations as well as by explicitly defining the repetitive and conditional steps of service behaviors.

We illustrate the enrichment of the behavior models by a simple example. Suppose that a service is defined by the means of its operations and their input and output messages

Example 1
A service description

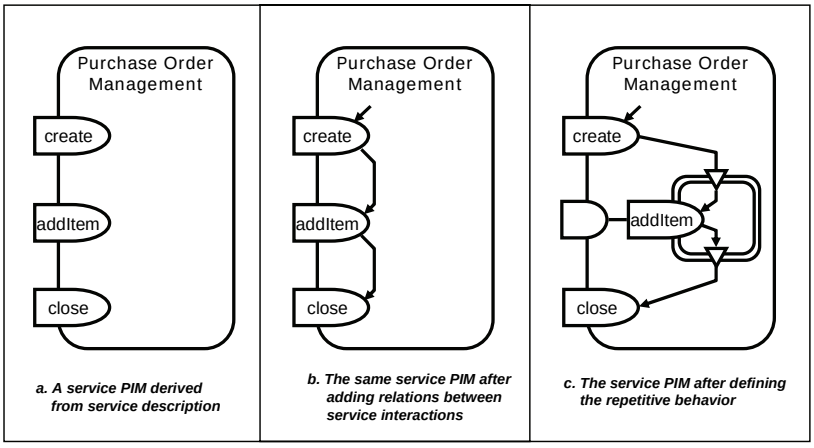
Service *PurchaseOrderManagementService*
Operation *CreateOrder*
InputMessage *CustomerInformation*
OutputMessage *OrderId*
Operation *AddItem*
InputMessage *OrderId*, *Item*
OutputMessage *Status* (e.g., available or not-available)
Operation *CloseOrder*
InputMessage *OrderId*
OutputMessage *Acknowledgment*

For the sake of simplicity, we have already abstracted from the technical details in the service description (data encoding schemata, IP addresses and transport protocols).

The service allows a customer to create a purchase order by providing customer information and to receive the id of the newly created order. Next, the customer may request a number of items and obtain their status (e.g., available or not available). If an item is available, it will be automatically added to the purchase order created in the previous step. Finally, the user may close the order and receive an acknowledgement.

Using the service description above we derive the platform-independent behavior model of the service (cf. Figure 5-74a). To make the behavior model of the service more precise we define the possible ordering of service operations (cf. Figure 5-74b) and explicitly define that the operation *AddItem* can be invoked multiple times before the order is closed (cf. Figure 5-74c).

Figure 5-74
Semantic enrichment of the behavior models



5.2.3 Step 3. Solving the Integration Problem at PIM level

In Step 3 (cf. Figure 5-75), we perform the design of the integration solution in a technology-independent manner, i.e., we define the platform-independent service model of the mediator.

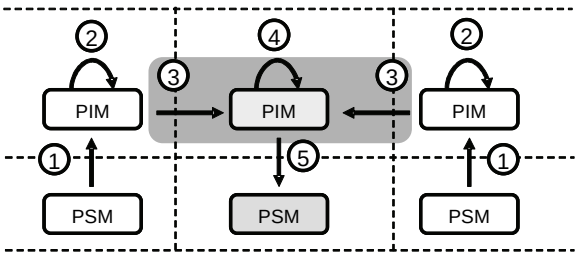


Figure 5-75
Step 3. Solving integration problem at PIM level

This step can be split into two parts: (i) the definition of the *information model* and (ii) the definition of the *behavior model* of the mediator. The purpose of the information model is to enable the compensation of the data mismatches by defining mappings between the elements of the information models of the systems to be integrated. The purpose of the behavior model is to enable the compensation of the mismatches in the interaction protocols by defining mappings between the operations of the requested and provided services of the systems to be integrated.

Definition of the information model of the mediator

The definition of the information model of the mediator is not different from that of an information model which consists of the logically related classes and properties from the information models of the systems to be integrated. The definition process consists of three steps, namely *discovery* of the relations among elements of the information models of the systems to be integrated, *representation* of these relations, and their *usage*.

The discovery step is either manual or (semi) automatic. In most cases, the relations among corresponding classes and properties of the information models of the systems to be integrated are discovered by interviewing business domain experts or interpreting domain standards. Therefore, it is important to represent the information models and the mapping relations in such a way that they can be understood and reviewed by domain experts. In addition to the manual mapping discovery, there are approaches that use heuristics and machine-learning techniques to discover similarities among elements of the information models and suggest mapping relations.

The semantic correspondence between elements of two information models can be expressed as a function of *subsumption*. Subsumption checking is the task of checking whether a concept *A* has more general

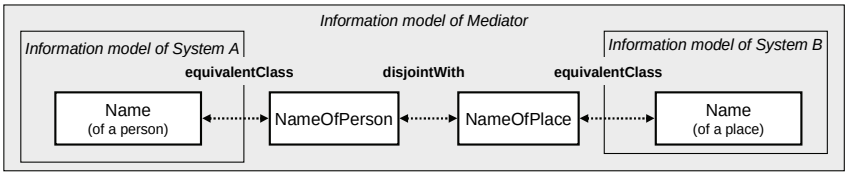
meaning than a concept *B*. In other words, subsumption is checking whether the criteria for being an individual of type *B* imply the criteria for being an individual of type *A*. Concept *A* is then denoted as *subsumer* and the concept *B* is denoted as *subsumee*. If *B* subsumes *A* and *A* subsumes *B*, then we can conclude that the concepts *A* and *B* are equivalent (i.e., they have the same meaning). In addition, two (or more) concepts can be checked to be disjoint. This is done by checking whether the logical conjunction of their membership criteria is subsumed by a concept that cannot have an individual (i.e., the empty concept).

In our approach, we define a number of mapping relations expressed as a function of subsumption. To state that a particular concept or property in one information model has the same meaning as a concept (or property) in a second information model we define the mapping relation *equivalentClass* (*equivalentProperty*, respectively). To state that a particular concept (or a property) in one information model has more specific meaning than a concept (or a property) in the second information model we use the relation *subClassOf* (*subPropertyOf*, respectively). Finally, to state that two concepts (or properties) have disjoint meanings we use the mapping relation *DisjointWith*. In the following we present how the mapping relations can be used to address the interoperability problems from Chapter 2.

Problem IP1. Different systems use the same symbol to represent concepts with disjoint meanings.

To address this problem we need to rename the symbol in the information model of the mediator and map the resulting (renamed) symbols using the relation disjointWith (cf. Figure 5-76).

Figure 5-76
Mapping symbols of disjoint concepts



Problem IP2. Different systems use the same symbol to represent concepts with overlapping meanings.

To address this problem we first specialise the concepts from the information models of both systems into two (or more) disjoint concepts. Next, we map the symbols of the corresponding specialised concepts using the *equivalentClass* relation (cf. Figure 5-77).

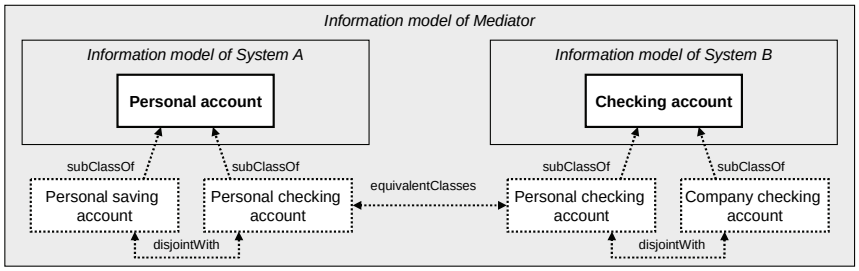


Figure 5-77 Mapping symbols of overlapping concepts (bidirectional)

Note that in some cases it is not necessary to specialise both corresponding concepts. For example, if we are sure that only the first system sends messages about "Personal account" then we will only need to specialise that concept to "Personal checking account". Instead of using *equivalentClasses* we then use the relation *subClassOf* (cf. Figure 5-77).

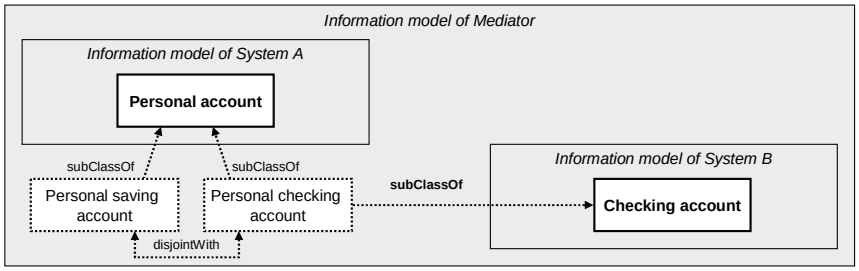


Figure 5-78 Mapping symbols of overlapping concepts (unidirectional)

Problem IP3. Different systems use the same symbol to represent concepts with more general (or more specific) meanings.

The solution of this problem is similar to the one of Problem IP2: we first specialise the concept from the information model of the first system into two (or more) disjoint concepts. Next, we map the symbol of the respective specialised concept to the corresponding symbol of the second system using *equivalentClass* relation (cf. Figure 5-79).

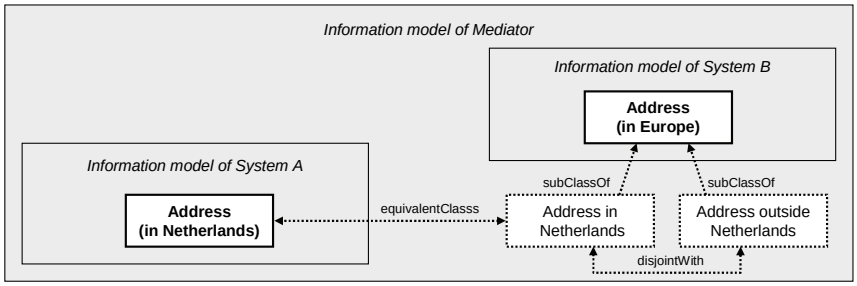
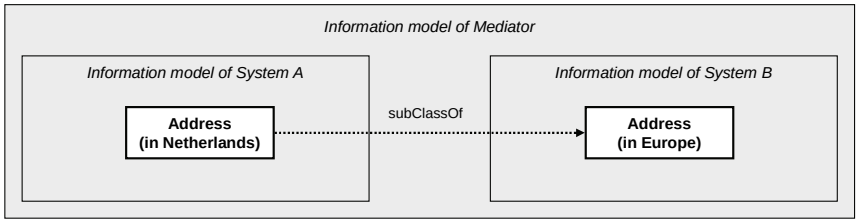


Figure 5-79 Mapping symbols of more specific/general concepts (bidirectional)

Similar to the solution of Problem IP2, if we are sure that only the first system sends messages about "Address (in Netherlands)" to the second system, we will simply map the corresponding symbols of the concepts using the relation *subClassOf* (cf. Figure 5-80).

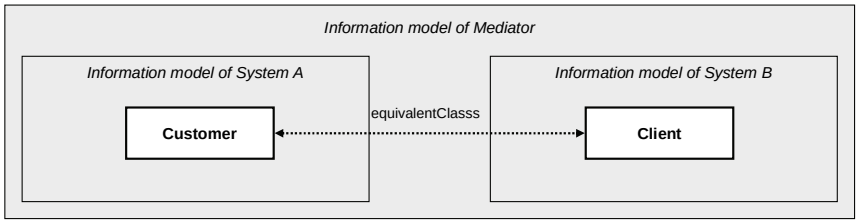
Figure 5-80
Mapping symbols of more specific/general concepts (unidirectional)



Problem IP4. Different systems use different symbols to represent the same concept.

To address this problem we simply use the mapping relation *equivalentClass* (cf. Figure 5-81).

Figure 5-81
Mapping symbols of equivalent concepts



Problem IP5. Different systems use different symbols to represent concepts with overlapping meanings.

The solution of this problem is identical as the one of Problem IP2: we first specialise the concepts from the information models of both systems into two (or more) disjoint concepts. Next, we map the symbols of the corresponding specialised concepts using *equivalentClass* relation (cf. Figure 5-82).

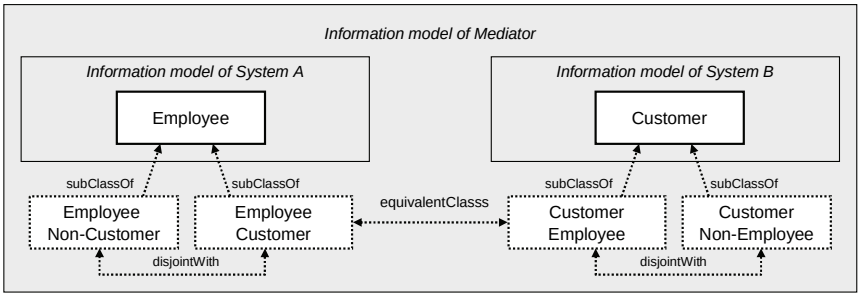


Figure 5-82 Mapping symbols of overlapping concepts (bidirectional)

Likewise, if we are sure that only the first system sends messages about "Employee" then we will only need to specialise that concept to "Employee Customer". Instead of using *equivalentClasses* we then use the relation *subClassOf* (cf. Figure 5-83).

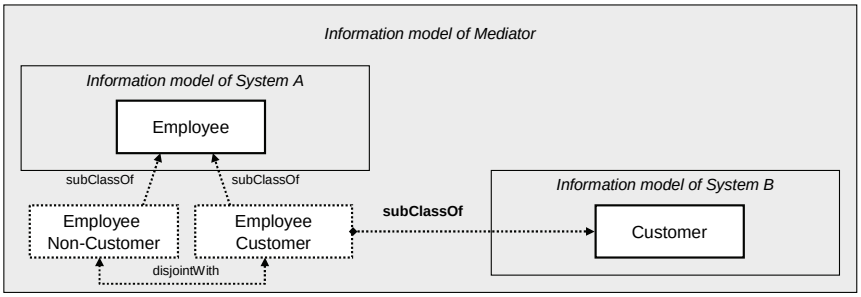


Figure 5-83 Mapping symbols of overlapping concepts (unidirectional)

Problem IP6. Different systems use the different symbols to represent concepts with more general (or more specific) meanings.

The solution of this problem is identical as the one of Problem IP3: we first specialise the concept from the information model of first system into two (or more) disjoint concepts. Next, we map the symbol of the respective specialised concept to the corresponding symbol of the second system using the *equivalentClass* relation (cf. Figure 5-84).

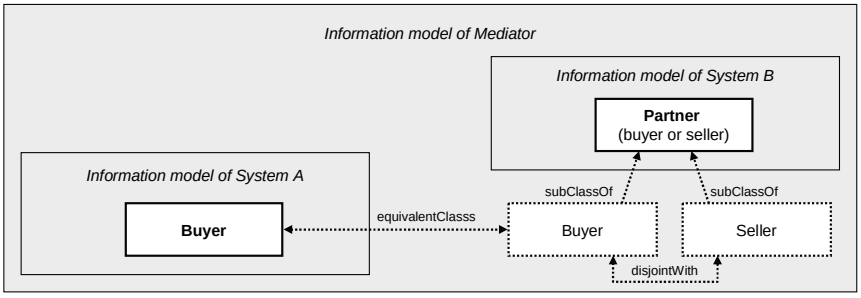
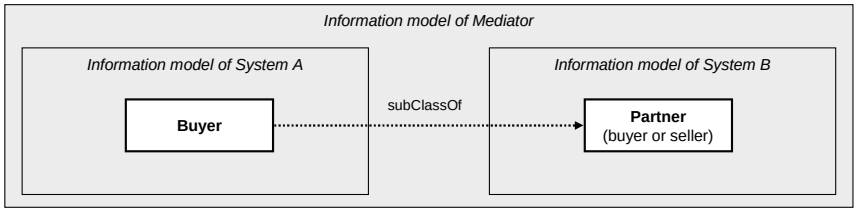


Figure 5-84 Mapping symbols of more specific/general concepts (bidirectional)

Similar to the solution of Problem IP3, if we are sure that only the first system sends messages about "Buyer" to the second system, we simply map the corresponding symbols of the concepts using the relation *subClassOf* (cf. Figure 5-85).

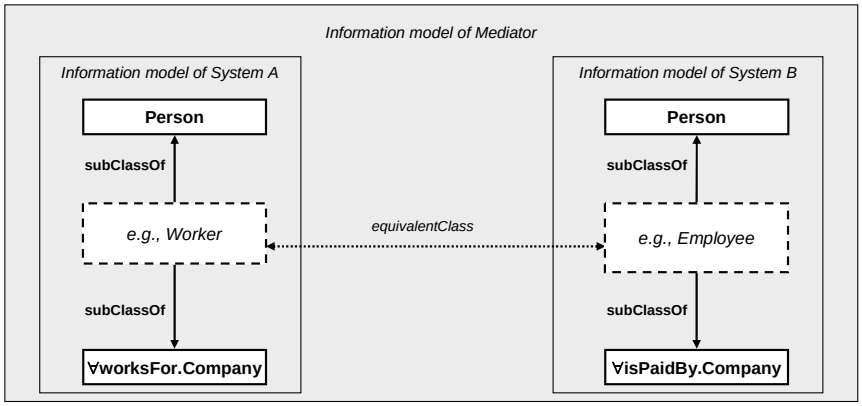
Figure 5-85 Mapping symbols of more specific/general concepts (unidirectional)



Problem IP7. Different definition of the same concept (also known as confounding conflicts).

To solution of this problem is no different than the solution of Problem IP4 - we simply use the mapping relations *equivalentClass* (cf. Figure 5-86).

Figure 5-86 Different definition of the same concept



The information mediation patterns are summarised in Figure 5-87.

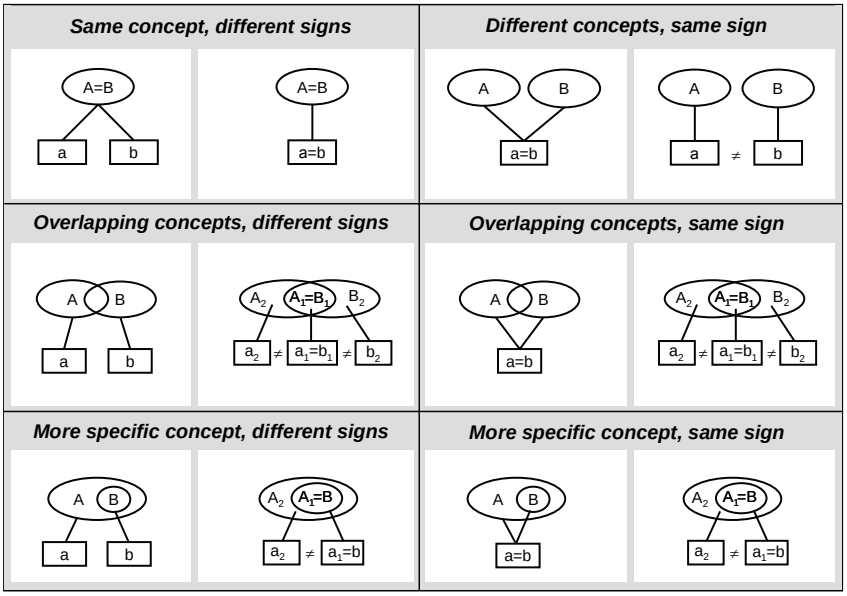


Figure 5-87 Summary of the solutions for the information problems

As said in Chapter 2, formal knowledge representation languages provide means for defining new classes and properties from existing ones. A new class can be defined as union, intersection or complement of other existing classes. In addition, a new (specialised) class can be defined by restricting the values of some property of an existing class. For example, class "Item" can be specialised to class "Available Item" by restricting the values of the property "status" to the value "available" (cf. Figure 5-88).

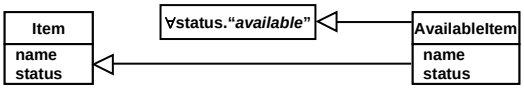


Figure 5-88 Specializing the class "Item" using its property "status"

As discussed earlier, the ability to define new (specialised) classes is crucial to enable the mapping of concepts with overlapping or more general/specific meaning.

Similar, formal knowledge representation languages provide means for defining new properties from existing ones. For example, a new property "partnerAddress" can be defined as a composition of the properties "location" and "address". Later, this property can be mapped to other property (e.g., "customerAddress") using the *equivalentProperty* relation (cf. Figure 5-89).

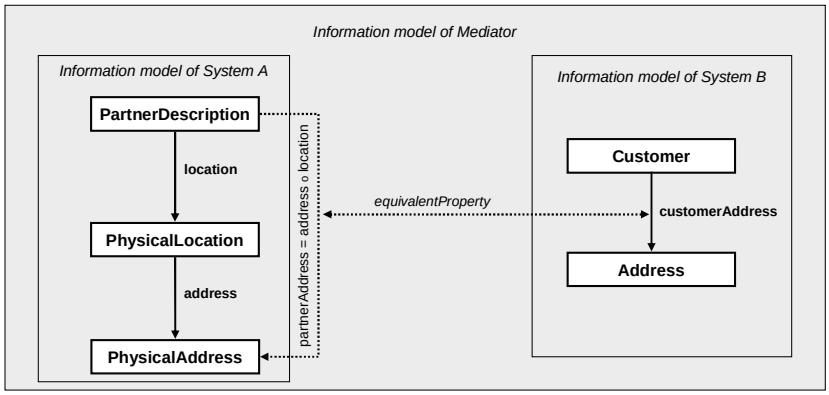


Figure 5-89
Example of composite property mapping

The use of a formal knowledge representation language allows for automatic discovery of the *equivalentClass*, *subClassOf* and *disjointWith* relations. This can be done by merging the information models of the systems to be integrated and testing each pair of concepts and properties for subsumption. For example, suppose that we have one information model defining classes "Business", "Person" and "SmallBusiness" as a "Business that employs at most 3 "Persons", and a second ontology introducing classes "Company", "Employee" and "SME" as a "Company" that has at most 10 employees.

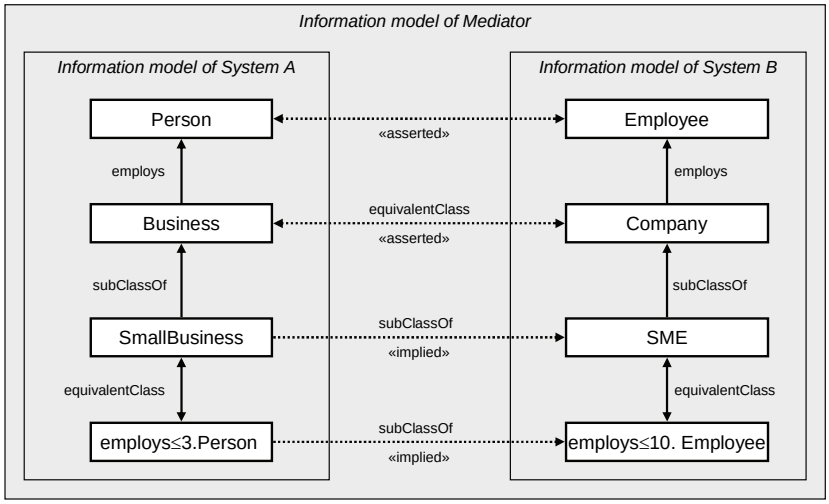


Figure 5-90
Example of composite property mapping

If we know that all employees are persons and we have already defined that *Business* is equivalent to *Company*, we can deduce that a *SmallBusiness* is a *SME*.

Definition of the behavior al model of the Mediator

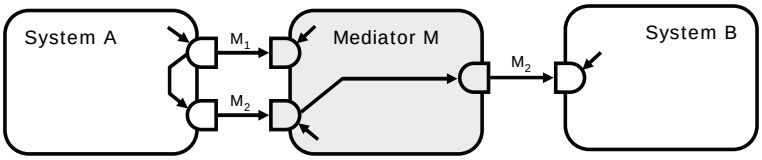
The behavioral problems can be associated with specific solutions for each problem. Based on these solutions the aim is to select only the relevant ones and compose them to form the behavior of the mediator. In our approach, we consider the partial solutions as simple behaviors and use them to compose more complex behaviors. The composition is recursively applied until the composed behavior solves the integration problem.

In Chapter 2, we identified possible behavior problems. In this section, we define solutions for these problems.

Problem BP1: Unexpected message. System A intends to send two messages, first M_1 and then M_2, whereas *System B* expects only message M_2.

To address this problem we define the mediation behavior M presented in Figure 5-91. *Mediator M* receives message M_1 and ignores it. Next, it receives message M_2 and sends it to *System B*.

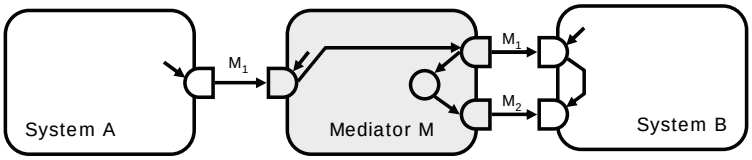
Figure 5-91
Unexpected message problem



Problem BP2: Insufficient message. System B expects two messages, M_1 and M_2, whereas *System A* intends to send only message M_2.

To address this problem we define the mediation behavior *M* presented in Figure 5-92. *Mediator M* receives message M_2 from *System A*. Next, it uses additional information (either provided by another system or derived from the execution history) to construct and send message M_1 to *System B*. Finally, the *Mediator* sends message M_2 to *system B*. Note, that this mismatch can only be compensated if *Mediator M* has all information necessary to construct message M_1.

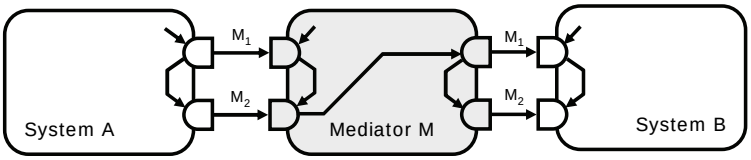
Figure 5-92
Insufficient message problem



Problem BP3: Message order. System A intends to send message M_1 first and then M_2, whereas *System B* expects first message M_2 and then M_1.

To address this problem we define the mediation behavior *M* presented in Figure 5-93. *Mediator M* receives first message M_1 and then message M_2. Next, it sends message M_2 first and then message M_1.

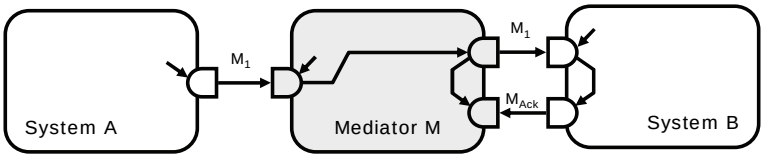
Figure 5-93
Message order problem



Problem BP4: Unexpected acknowledgement. System A sends message M_1 to *System B* and continues without expecting an acknowledgement, whereas *System B* intends to send message M_{ack} to acknowledge the reception of message M_1.

To address this problem we define the mediation behavior *M* presented in Figure 5-94. *Mediator M* receives message M_1 from *System A*, sends it to *System B*, and then receives the acknowledgement M_{ack} on behalf of *System A*.

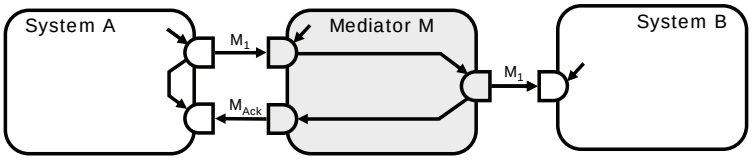
Figure 5-94
Unexpected acknowledgement problem



Problem BP5: Insufficient acknowledgement. System A sends message M_1 and expects acknowledgement M_{ack} whereas *System B* does not intend to send such an acknowledgement.

To address this problem we define the mediation behavior *M* presented in Figure 5-95. *Mediator M* receives message M_1, sends it to *System B*, and then sends an acknowledgement (M_{ack}) to *System A* on behalf of *System B*.

Figure 5-95
Insufficient acknowledgement problem

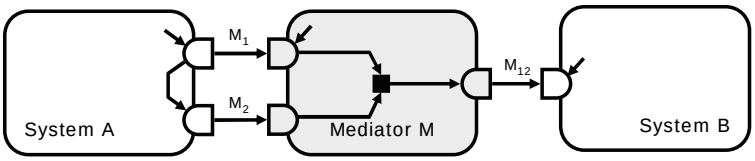


Problem BP6: Message aggregation. System A sends messages M_1 and M_2 whereas *System B* expects one message M_{12} that aggregates M_1 and M_2.

To address this problem we define the mediation behavior *M* presented in Figure 5-96. *Mediator M* receives both messages M_1 and M_2. Then it uses the

information from these two messages to construct M_{12}. Finally, the mediator sends M_{12} to *System B*.

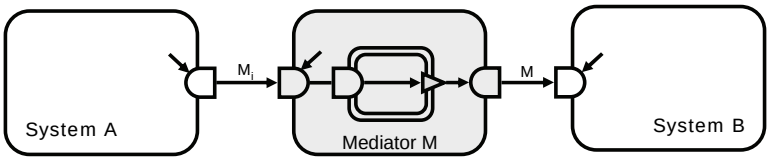
Figure 5-96 Message aggregation



Problem BP7: Aggregation of multiple messages of the same type. *System A* sends message M_i n times whereas *System B* expects one single message M that aggregates all *n* messages M_i.

To address this problem we define the mediation behavior *M* presented in Figure 5-97. *Mediator M* starts a process of receiving messages M_i until some condition evaluates to true. Next, it uses the information in the received messages to construct M and then sends M to *System B*.

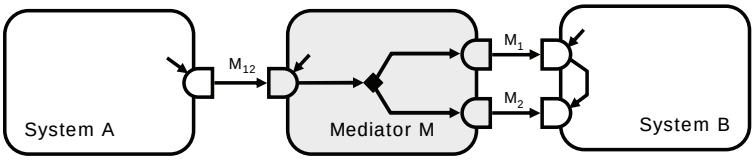
Figure 5-97 Aggregation of multiple message s of the same type



Problem BP8: Message splitting. *System B* expects two messages M_1 and M_2 whereas *System A* intends to send only one message M_{12} that contains both M_1 and M_2.

To address this problem we define the mediation behavior *M* presented in Figure 5-98. First, *Mediator M* receives message M_{12}. Then, it constructs M_1 and M_2 using the information from M_{12}. Finally, the mediator sends M_1 and M_2 in the order expected by *System B*.

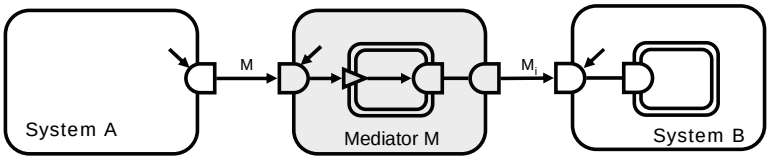
Figure 5-98 Message splitting problem



Problem BP9: Splitting to multiple messages of the same type. *System B* expects n times message M_i whereas *System A* intends to send only one message M that contains all *n* messages M_i.

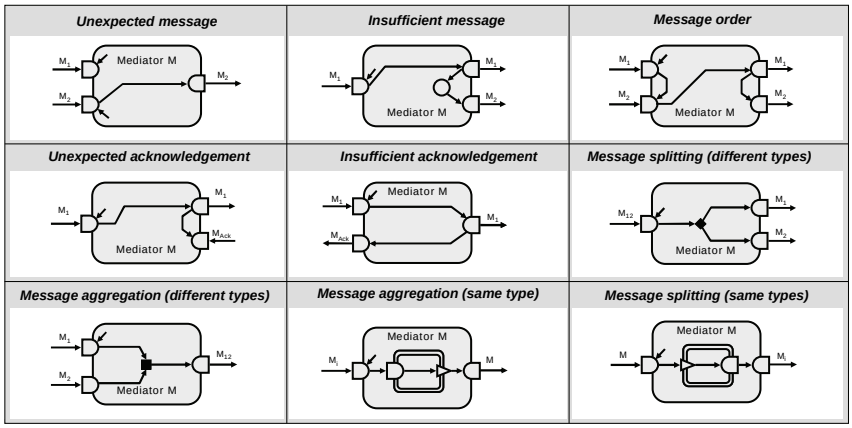
To address this problem we define the mediation behavior *M* presented in Figure 5-99. *Mediator M* first receives message M. Then it starts a process of constructing M_i from the information in M and sending M_i to *System B*. This process is repeated until some condition evaluates to true.

Figure 5-99
Splitting to multiple messages of the same type



The behavior mediation patterns are summarised in Figure 5-100.

Figure 5-100
Summary of the solutions for the behavior problems



The definition of the behavior model of the *Mediator* requires the definition of the services (i) provided and (ii) requested by the *Mediator* and the composition of these services by relating their operations. This is done by inspecting the mapping relation in the information model of the *Mediator* and defining relations among respective interaction contributions. For example, suppose that *System A* sends a message M_1 that contains the values of the properties name and address. Suppose that *System B* expects two messages M_2 and M_3 that contain the elements *customerName* and *customerAddress* respectively (cf. Figure 5-106).

5.2.4 Step 4. Verification of the Integration Solution

This step (cf. Figure 5-101) of our integration method analyses whether the proposed integration solution really enables interoperability between the systems to be integrated.

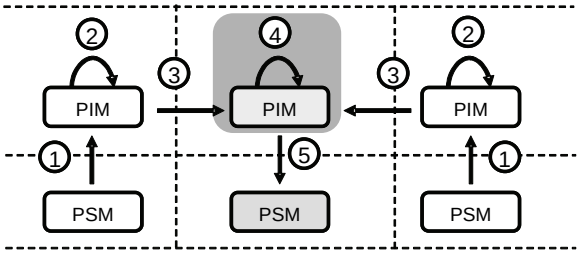


Figure 5-101
Step 4. Verification of the integration solution

For that reason, we use the necessary conditions identified in Section 5.1. First, we check whether all defined interactions between the mediator and the systems to be integrated can establish results. Next, we verify whether the integrated behavior of the mediator and the systems to be integrated can be performed. The verification method is omitted here and explained in detail in Section 5.3.

5.2.5 Step 5. Deriving the PSM of the Integration Solution

In this final step (cf. Figure 5-102), the platform-independent service model of the mediator is transformed into a platform-specific model in terms of some implementation technology.

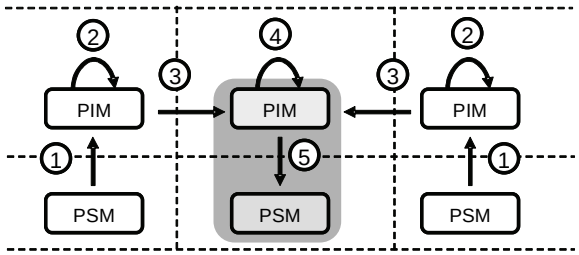


Figure 5-102
Step 5. Deriving the PSM of the integration solution

In our approach, we do not assume a particular execution platform. For example, the platform-specific service model of the mediator can be transformed to a WS-BPEL specification, EJB, or .Net application. In this section, we only present an abstract architecture of possible execution platforms (cf. Figure 5-103).

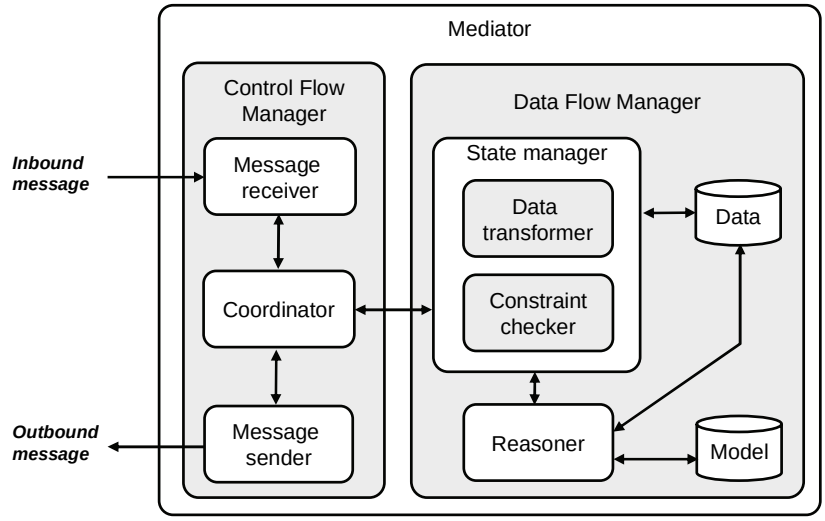


Figure 5-103 Abstract architecture of the *Mediator*

The architecture of the *Mediator* consists of two main components – *Control Flow Manager* and *Data Flow Manager*. The *Control Flow Manager* is responsible for sending and receiving messages in a particular order as well as for querying and updating the state of the *Mediator*. The *Data Flow Manager* in turn, is responsible for managing the state of the *Mediator* and for performing the necessary data transformations and constraint checking.

The *Control Flow Manager* consists of three subcomponents – *Message receiver*, *Message sender* and *Coordinator*. The *Message receiver* is responsible for receiving all inbound messages and the *Message sender* for sending all outbound messages. The *Coordinator* executes the behavior specified in the behavior al model of the *Mediator*, i.e., based on the current state it activates and deactivates the *Message receiver* and *Message sender*. When a message is received, the *Coordinator* interacts with the *Data Flow Manager* to update the state of the *Mediator*. When a message is to be sent, the *Coordinator* interacts with the *Data Flow Manager* to obtain the data required to construct the outbound message.

To derive the *Control Flow Manager* we use the approach described in (Dirgahayu, 2007). The proposed approach is divided into three steps: *pattern recognition*, *pattern realization* and *activity transformation (cf.* Figure 5-104*)*.

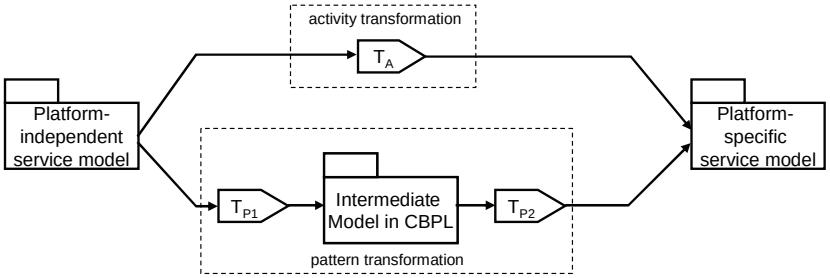


Figure 5-104 Transforming the platform-independent service model of the mediator to a platform-specific model

To decouple the pattern recognition and pattern realization (and this way provide support for building reusable transformations), the authors define *Common Behavior al Patterns Language* (*CBPL*). Each CBPL pattern is represented as *sequence*, *concurrence*, *selection* and *iteration* (cf. Figure 5-105). A *sequence* contains one or more activities to be executed in succession. A *concurrence* contains two or more activities that can be executed independently. An *iteration* contains an activity to be executed repeatedly as long as its condition holds. A *selection* contains one or more *cases* to be selected. A *case* contains an activity to be executed when its condition holds. The basic CBPL patterns are illustrated in Figure 5-105.

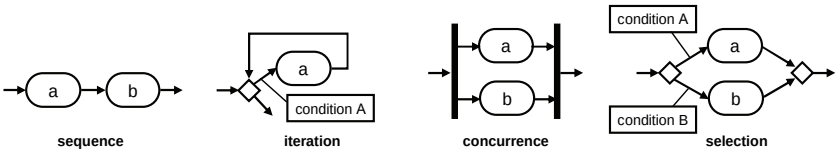


Figure 5-105 Sequence, concurrence, selection and iteration

The *Data Flow Manager* consists of two components – *State manager* and *Reasoner*. The *State manager* is responsible for updating the state of the *Mediator* (after receiving a message) and for querying that state (before sending a message or when checking a constraint). In some cases, data in the received message may have to be transformed before updating the state. For that purpose the *State manager* uses the *Data transformer* component. Likewise, in some cases the *State manager* uses the *Data transformer* to construct new messages. The *Data transformer* is in fact the component that implements the mapping relations specified in the information model of the mediator. Similar to *Data transformer*, the *Constraint checker* queries the state of the mediator and provides an answer whether a constraint holds or not.

To take full advantage of the formal specification of the information model of the *Mediator* the *Data Flow Manager* may contain a *Reasoner* component. The *Reasoner* uses the formal knowledge specified in the information model of the *Mediator* in conjunction with the facts about the current state of the *Mediator* to infer new state information, i.e., it makes all *implicit* knowledge about the state more *explicit*. In addition, the *Reasoner* can

be used by the *Data transformer* and the *Constraint checker* as an intelligent query engine and constraint solver.

A platform-specific service model contains information that is not present in the platform-independent service model. Examples of such information are the XML namespaces of the exchanged messages or the types of the service operation (e.g., in case of WS-BPEL - *invoke*, *receive* or *reply*). To provide the required platform-specific information we annotate the elements of the platform-specific service model.

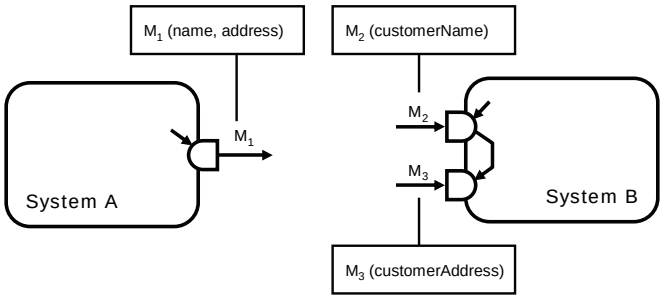


Figure 5-106 Example of two systems to be integrated

First, we define a *Mediator* behavior that matches the receiving of M_1 and sending of M_2 and M_3. In addition, we add the relation between M_1 and M_2.

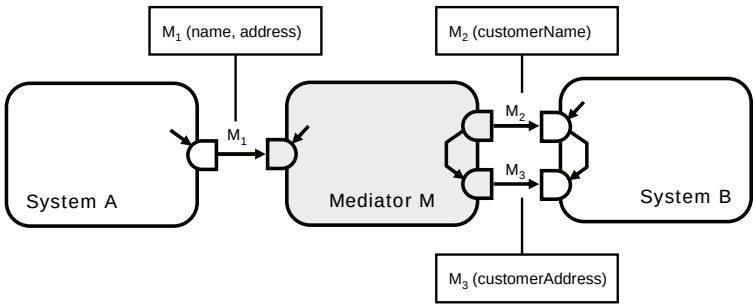


Figure 5-107 Defining a mediator that matches the send and received messages by the systems to be integrated

Suppose that the information model contains the mappings

$$\forall x{:}\exists \text{name}(M_1, x) \wedge \ \forall y{:}\exists \text{customerName}(M_2, y) \rightarrow y = x$$

$$\forall x{:}\exists \text{address}(M_1, x) \wedge \forall y{:}\exists \text{customerAddress}(M_3, y) \rightarrow y = x$$

Analyzing the mappings, we discover that we need information from M_1 to construct M_2 and M_3. Therefore, we add message-splitting behavior to the *Mediator* (cf. Figure 5-108)

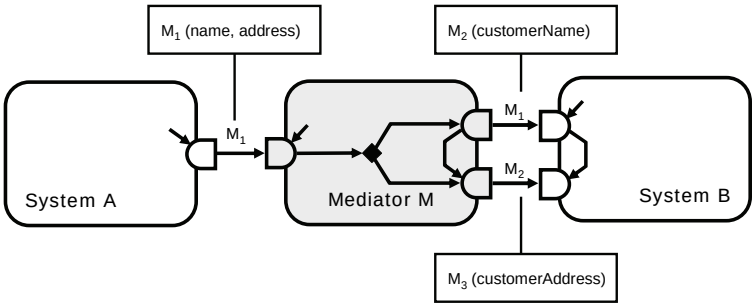


Figure 5-108 Adding message splitting behavior to the mediator

In some cases, the information mappings are not sufficient to define the complete behavior of the *Mediator*. For example, different messages could provide the same type of information used to produce another message. In such cases the designer of the *Mediator* has to decide which input message to use. In some other cases, the behavior model of the *Mediator* can define custom processing logic. For example, the *Mediator* could wait for a message from some system for a certain period of time. If it does not receive such a message, it has to send a timeout message to another system. Such a custom processing logic has to be defined manually.

5.3 Method for Formal Verification of System Interoperability

After defining the information and behavior al model of the *Mediator* (step 3) we use the identified requirements in Section 5.1 to verify whether the mediator and the systems to be integrated form an interoperable integrated system (step 4).

Necessary conditions 1 and 2 can be verified by *satisfiability reasoning*. *Satisfiability* reasoning is a special case of subsumption reasoning. In this case the subsumer is the empty class (i.e., a class that cannot have any instances). If a concept *A* is subsumed by the empty class, we will say that the class *A* is unsatisfiable. This means that no individual can be of type *A*.

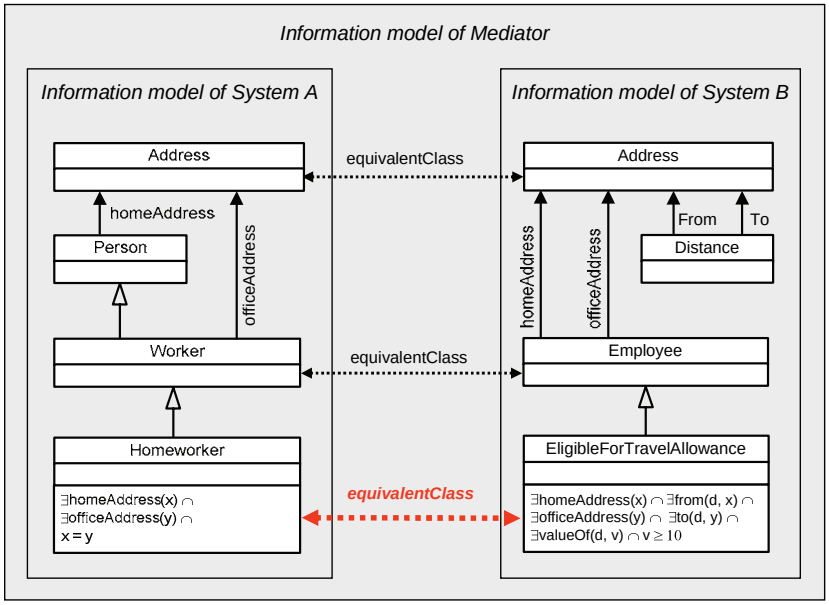
5.3.1 Verifying Condition 1

To verify the Condition 1 we check the *consistency* of the information model of the mediator. We define a model to be consistent if all classes and mappings in that model are satisfiable. To illustrate an inconsistent model we use the following example (cf. Figure 5-109).

Suppose that the information model of *System A* defines the class "Homeworker" as subclass of "Worker" whose home address is the same as

its office address. Suppose that *System B* defines the class "EligibleForTravelAllowance" as a subclass of "Employee" whose home address is at least 20 km from his office address. In addition, suppose that there exist a fact defining that the distance between two same addresses is 0. When a system integrator defines *equivalentClass* mapping between the classes "Homeworker" and "EligibleForTravelAllowance" the model of the mediator becomes inconsistent. This is because the mapped classes become unsatisfiable, i.e., it is not possible to have distance between the home and office address 0 and at least 20 at the same time. Using a reasoner such an inconsistency can be immediately discovered and the system integrator can be warned.

Figure 5-109 Verifying Necessary Condition 1



5.3.2 Verifying Condition 2

To verify Condition 2, we define a class that represents the possible results of an interaction and check whether this class is satisfiable. The class of possible results of an interaction is defined as the conjunction of the constraints on the results of all participating interaction contributions. To illustrate an unsatisfiable class we use the following example (cf. Figure 5-110).

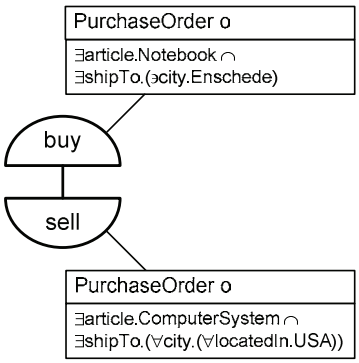


Figure 5-110 Verifying Necessary Condition 2

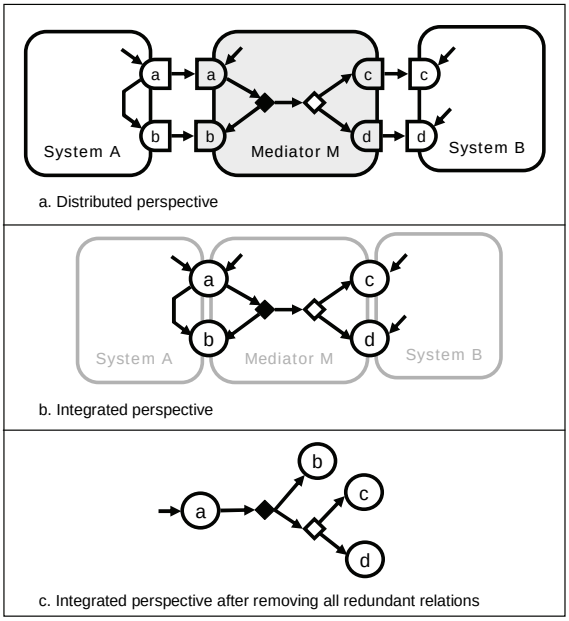
Suppose that the service model of *System A* defines an interaction contribution *buy* which establishes a Purchase order containing article of type Notebook and shipping address in the city Enschede. Suppose that the service model of *System B* defines an interaction contribution *sell* which establishes a purchase order containing article of type Computer system and a shipping address to any city in the USA. Suppose that there exist facts defining that USA and Europe are disjoint classes, Enschede is located in the Netherlands and Netherlands is located in Europe. The interaction *sell-buy* cannot happen because its result (defined as a logical conjunction of the constraints of the participating systems) is unsatisfiable. This is because the one system requires shipping address in USA, the other system requires shipping address in Europe and USA and Europe are disjoint classes, i.e., they cannot share an address. Using a reasoner such unsatisfiability can be immediately discovered and the system integrator can be warned.

5.3.3 Verifying Condition 3

To verify Condition 3, we need to check whether the integrated service behavior satisfies the constraints of the participating systems on the possible ordering of interactions among them. For that purpose, we first abstract from the participation of each systems in a service and construct the behavior of the integrated system. This is an operation supported by the COSMO framework. Next, we abstract the repetitive behaviors to non-repetitive behaviors, i.e., we only consider one iteration of each repetitive behavior. Finally, we transform the integrated behavior to a formalism that allows for constructing the state space of the integrated system and performing reachability analysis. Reachability analysis is deciding whether or not there exists a path from a distinguished node s to a distinguished node t in a directed graph. In our case, the nodes in the graph represent the possible states of the integrated system (i.e., results established in interactions among systems), and the directed arcs between nodes represent

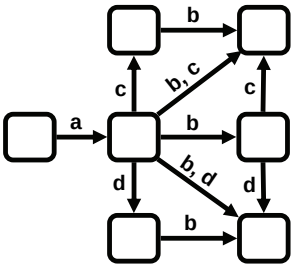
the order of the establishments of these results. The steps of the verification method are illustrated in Figure 5-111 and Figure 5-112.

Figure 5-111
Verifying Necessary Condition 3



The resulting state space graph of the behavior presented in Figure 5-111 is presented in Figure 5-112.

Figure 5-112
State space of the behavior in Figure 5-111



We can perform reachability analysis by querying the graph. For example, we can check whether the results of a, b and d can be established in the order $a \rightarrow b \rightarrow d$.

Suppose that we modify the example from Figure 5-111 by adding a constraint in the behavior of *System B*, defining that d can only occur after c has occurred (cf. Figure 5-113).

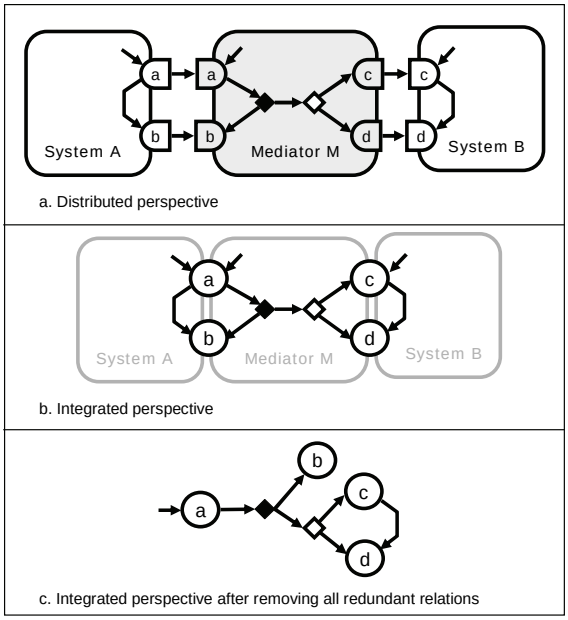


Figure 5-113 Verifying Necessary Condition 3 after modifying the behavior of *System B*

The state space graph of the integrated behavior from Figure 5-113 is depicted in Figure 5-114.

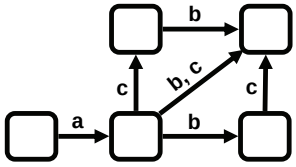


Figure 5-114 State space of the behavior in Figure 5-113

As we can see, the execution scenario $a \rightarrow b \rightarrow d$ is no longer possible. In fact, d can never occur since it depends on both the occurrence and non-occurrence of c.

In the table, presented in Figure 5-115, we summarise the necessary conditions and the methods to verify them.

Title	Description	Verified by
Necessary condition 1	*A necessary condition for semantic interoperability of two systems is that the sent message x is a valid instance of S, the received message y is a valid instance of T and the sent and the received messages together satisfy the predicates defined by the source-to-target mapping M.*	*Checking the consistency of the information model of the integration solution*
Necessary condition 2	*A necessary condition for pragmatic interoperability of an interaction is that at least one result that satisfies the constraints of all contributing systems can be established.*	*Checking the satisfiability of the class defining the possible results of the interaction*
Necessary condition 3	*A necessary condition for pragmatic interoperability of a service is the existence of at least one execution scenario that can establish all required results.*	*Constructing the state space of the integrated system and performing a reachability analysis*

Figure 5-115
The summary of the necessary conditions and the methods to verify them

5.4 Related Work

The *Web Services Modeling Ontology* (WSMO) consists of four main components – *ontologies*, *goals*, *web services* and *mediators*. Data mediation is achieved through the design and implementation of adapters specifying mapping rules between ontologies. During runtime, the approach considers specific mediator services, which perform data transformations at instance level. The mediator interaction behavior is described by means of *Abstract State Machines* (AST), consisting of *states* and *guarded transitions*. A *state* is described within an ontology and the *guarded transitions* are used to express changes of states by means of transition rules. However, this implicit behavior specification may be neither intuitive nor trivial to make sure that the expectations implied by the designed transition rules match the expected operation message exchange patterns.

The *jABC approach* (Steffen, 2006) uses *Service Logic Graphs* as choreography models, allowing the designer to model the mediator graphically, in high-level modeling language by combining reusable building blocks into (flow-)graph structures. These basic building blocks are called *Service Independent Building Blocks* (SIB) and have one or more edges (branches), which depend on the different outcomes of the execution of the functionality represented by the SIB. The provided model-driven design tools allow the modeling of the mediator in a graphical high-level modeling language and support the derivation of an executable mediator from these models. More recently (Margaria, 2008), the approach has focused on how to apply a tableau-based software composition technique to generate automatically the mediator's interaction behavior. This uses a *Linear Time Logic* (LTL) planning algorithm originally embedded in the jABC platform. However, the applicability of automated synthesis of the mediator's

business logic is still limited considering the kind of assumptions being made. In comparison with the jABC approach, our approach does not cover automated synthesis of the mediator logic as it intentionally leaves the planning task to the business domain experts.

The core concept of the *FOKUS* (Barnickel, 2008) approach is the integration of ontology mappings into WS-BPEL processes. The approach addresses the data mediation by applying *semantic bridges* to mediate between different information models and representations. Semantic bridges are described as a set of description logic-based axioms relating entities in business information models that are defined in different ontologies but have a similar meaning. The description logic-based data model provided by ontologies in conjunction with semantic bridges allows for applying automatic semantic matching and reasoning mechanisms based on polymorph representations of service parameters. The interaction behavior of the mediator has been manually designed and addressed by using a WS-BPEL engine as the coordinating entity. Some WS-BPEL enhancements were developed to integrate semantic bridges and to support data flow specifications in terms of rules. These enhancements were implemented as external functions that can be plugged into WS-BPEL engines. Thus, in contrast to our approach, the presented approach designs the mediation solution at technology level. It relies strongly on WS-BPEL and cannot easily be used with alternative technologies.

5.5 Conclusions

In this chapter, we presented a method for the semantic integration of service-oriented applications. The key feature of the proposed method is that semantically enriched service models are employed at different levels of abstraction to develop end-to-end integration solutions from business requirements to software realization. This way, the integration problem is solved at a higher level of abstraction by business domain experts and then (semi-)automatically transformed to a software solution by adding technical details by the IT experts.

First, using the interoperability levels identified in Chapter 2, we answered *Research question Q4*: "What is necessary for two or more systems to interoperate? How can we formally check if two or more systems are interoperable?". Second, by providing integration method, we answered *Research question Q5:* " How can two or more non-interoperable systems be integrated and how can such integration be achieved in a systematic way? Does such integration solve the drawbacks of existing integration approaches?". Our method uses the interoperability problems identified in Chapter 2 and provides solutions for each of them. In addition, the method

defines all steps for building end-to-end integration solutions from business requirements to software realization.

To achieve our objectives we used MDA and formal KR.

KR provides mechanisms to define new classes by restricting properties of existing ones. This allows service integrators to refine the information models of the systems to be integrated and solve integration problems related to concepts with overlapping (or more general/specific) meaning. Second, KR provides mapping relations to express equivalence, subsumption and disjointness between concepts. These mapping relations become part of the information model of the mediator which allows this model to be automatically checked for consistency. As we showed earlier, checking model consistency is required to verify the necessary condition 1. Third, KR provides mechanism to define new classes as intersection, union or complement of existing classes. Such defined classes can be automatically checked for satisfiability. As we showed earlier, this is required to verify condition 2. Fourth, when using KR in combination with a reasoner the system integrators can get immediate feedback whether a model is still consistent after adding new information to it as a part of the semantic enrichment step. Finally, KR provides means for run-time data transformations. Class definitions can be used to define source-to-target mappings and can be executed at run-time by a reasoner to transform data in the messages exchanged among the systems to be integrated.

Similar to KR, MDA also plays an important role in our approach. First, MDA provides a means to insulate integration solutions at business level from the implementation technologies. In this way, the same business integration solution can be reused to implement different IT integration solutions using different implementation technologies. In addition, when the implementation technology evolves, the same business integration solution can be reused to generate IT integration solution that takes advantages of the new features of that technology. Second, MDA provides a means to define domain-specific languages (DSLs). This way, business domain experts can solve integration problems using concepts that are closer to their domain. In addition, a DSL shields business experts from the formal knowledge representation techniques and the syntax of data transformation definitions while at the same time enable them to analyse, verify and optimise their integration solutions and discover possible problems at an early stage. Finally, MDA consists of standards and best practices across a range of software engineering disciplines. Because of the standardization, there are already many tools available.

In Chapter 6 and 7, we validate our integration and verification methods in two case studies.

Part IV.
Validation

Chapter 6

Validation Goal and Claims

The main contribution of this thesis is the provisioning of a *method for the semantic integration of service-oriented applications*. The key feature of the proposed method is that *semantically enriched service models* are be employed at *different levels of abstraction* to develop *flexible*, end-to-end integration solutions from business requirements to software realization.

In Chapter 1, we identified *requirements* for integration methods in general. To validate whether or not the solution proposed in Chapter 5 meets these requirements, in the following, we make a number of *claims* and provide *arguments for their validity*. This is done by applying our integration method in a particular context (i.e., solving two characteristic integration problems presented in Chapter 7 and 8, respectively). When applying our integration method we observed a number of effects. Chapter 9 analyses these observations and argues to what extent our integration method meets the requirements defined in Chapter 1. In particular, our analysis provides more insight into the following issues:

- What effects did we observe when applying our integration method in a particular context?

- Did these observations show that the proposed integration method really meets the requirements defined in Chapter 1? To what extent?

- How similar are the presented cases? Can we generalise the observations made to a broader context? What did we learn when applying our integration method to the cases presented in Chapter 7 and 8 that is relevant for other cases?

In the following, we present the validation claims. Each claim corresponds to a particular requirement.

Claim 1: Service PIMs can be derived from their respective PSMs. This claim is to validate whether our method meets *Requirement R1*. We verify it in Chapter 7 by providing a concrete model transformation that abstracts a service specification in terms of WSDL to service PIMs in terms of COSMO. In addition, Chapter 8 shows how a service PIM can be obtained if no explicit service PSM is available.

Claim 2: COSMO provides all required concepts to model platform-independent integration solutions. This claim is to validate whether our method meets *Requirement R1*. We verify it in Chapter 7 and 8 by specifying the PIM of the integration solution using COSMO. In Chapter 7, we focus on the concepts for modeling *service aspects* whereas in Chapter 8 we also show the usefulness of COMSO *perspectives* and *abstraction levels*.

Claim 3: The service models of the systems to be integrated can be semantically enriched. This claim is to validate whether our method meets *Requirement R2*. We verify it in Chapter 7 by defining additional service interactions and causality relations in the behavior models of the systems to be integrated. In addition, we demonstrate how an information model can be mapped to a domain-specific ontology, namely *Universal Data Element Framework* (*UDEF* (UDEF, 2006)).

Claim 4: The necessary conditions for interoperability can be formally checked. This claim is to validate whether our method meets *Requirement R3*. We verify it in Chapter 7 and 8 by providing concrete mappings from COSMO to OWL and Petri Nets and verifying the necessary conditions for interoperability (defined in Chapter 5) using logical reasoners.

Claim 5: The same solution PIM can be used to derive different solution PSMs. This claim is to validate whether our method meets *Requirement R4*. To verify it, we present a hypothetic scenario in which the requirements for the implementation technology of the case, presented in Chapter 8, change. To address the new requirements we specify a new model transformation from PIM to PSM and show how the *same service PIM* of the integration solution can be reused to *derive new service PSM* (in terms of the new technology).

Claim 6. The same model transformations can be used to solve different integration problems. This claim is to validate whether our method meets *Requirement R5*. To verify the claim, we present a variation of the case, presented in Chapter 7, in which the business requirements change. To address the new business requirements we *update* the service PIM of the existing integration solution and then *reuse* the same model transformations to derive the new service PSM.

The table below presents the correspondence between method properties and claims. In addition, we give references to sections, in which the claims are validated.

Requirement	***Claim***	***Validated in***
Requirement R1: The method should provide for defining the integration solutions in terms of the problem domain, rather than in terms of solution technologies.	*Claim C1: Service PIMs can be derived from their respective PSMs.*	*Chapter 7, Section 7.2.1 and 7.3.1, and Chapter 8, Section 8.2.1*
	Claim C2: COSMO provides all required concepts to model platform-independent integration solutions.	*Chapter 7, Section 7.2.3 and 7.3.3, and Chapter 8, Section 8.2.2*
Requirement R2: The integration method should enable the semantic integration of services.	*Claim C3: The service models of the systems to be integrated can be semantically enriched.*	*Chapter 7, Section 7.2.2, and Chapter 8, Section 8.2.1*
Requirement R3: The integration method should enable the formal verification of the integration solution.	*Claim C4: The necessary conditions for interoperability can be formally checked.*	*Chapter 7, Section 7.2.4 and 7.3.4, and Chapter 8, Section 8.2.3*
Requirement R4: The integration method should allow for changes in the implementation technology.	*Claim C5: The same solution PIM can be used to derive different solution PSMs.*	*Chapter 8, Section 8.3*
Requirement R5: The integration method should allow for changes of the business requirements.	*Claim C6. The same model transformations can be used to solve different integration problems.*	*Chapter 7, Section 7.3*

The remainder of this validation part of the thesis is organised as follows: In Chapter 7 we apply our integration method to solve a reference integration problem, defined in the Semantic Web Service Challenge (SWSC). SWSC provides an infrastructure for testing semantic web service technologies and a forum for discussion based on a common application base. In Chapter 8, we apply our integration method to solve a real-world integration problem

from the travel domain. More precisely, we perform a lab experiment, i.e., we apply our integration method using real-world data in a lab setting. Finally, Chapter 9 analyses the cases presented in Chapter 7 and 8, and identifies similarities and differences. By doing this, we want to provide further insight into the applicability of our integration method in more general context. In addition, we present a number of challenges we faced when solving the integration problems in Chapter 7 and 8. Finally, we summarise important lessons learnt.

Chapter 7

The Semantic Web Service Challenge Case

In this chapter, we validate our integration method presented in Chapter 5. The chapter is organised as follows: In Section 7.1, we introduce the Semantic Web Service Challenge initiative. Section 7.2 applies our integration method to solve a reference integration problem described in the Scenario 1 of SWSC. Section 7.3 demonstrates how our method supports the change of integration requirements by applying it to an integration problem described in the Scenario 2 of SWSC. Finally, in Section 7.4, we provide a short summary of the presented case. A detailed discussion about the applicability of our integration method is presented in Chapter 9.

7.1 The Semantic Web Service Challenge

To validate our integration method we use a reference integration problem, defined in the *Semantic Web Service Challenge* (SWSC)[33]. SWSC provides and infrastructure for testing semantic web service technologies and a discussion forum based on a common application base. In a series of workshops, the participants have to solve a set of integration problems using their methods and tools. For example, the participants must build software that invokes the right web services with the right sequence of correct messages in order to satisfy a certain scenario.

The scenarios are described in a Wiki provided by the SWSC. Corresponding to each scenario there is a set of working web services that SWSC participants can access. The SWSC organisers maintain these services and evaluate whether a participant has "passed" a scenario problem

[33] http://sws-challenge.org

or sub-problem by examining the log of web service messages exchanged, or by using dedicated software.

In addition to the provided infrastructure, SWSC organises a series of workshops. Workshops are held to provide consensus verification and to evaluate claims made by the participants. In these workshops, the participants present papers, that went through a review process, in which they present their claims. The organisers of SWSC verify whether or not the claimed problems have actually been solved. Finally, the workshop participants, either in teams or as a whole, evaluate claims of ease of response to a problem change by evaluating the actual code. Evaluation results are publicly posted on the website and certified by the consensus of the workshop in which they were made.

The SWSC evaluation methodology is evolving in the W3C Semantic *Web Services (SWS) Testbed Incubator*[34]. The mission of the W3C SWS Testbed Incubator Group is to develop a standard methodology for evaluating semantic web services based upon a standard set of problems as well as to develop a public repository of such problems.

Currently the SWSC includes members from both academia (Stanford University, University of Postdam, University of Stuttgartt, University of Southampton, University of Georgia, University of Karlsruhe, Politecnico di Milano, Trinity College, Universidad Politecnica de Madrid, University of Bolzano, Open University UK, University of Trento, Italy, Ulm University, Free University Berlin, and STI Innsbruck) and from the industry (HP Palo Alto USA, SAP Germany, IBM Research USA, Semagix, USA).

7.2 Scenario 1

To validate our integration method we start with Scenario 1 defined in the SWSC.

A manufacturing company called *Moon* uses two back-end systems to manage its order processing: a *Customer Relation Management (CRM)* and a *Stock Management (SM)* system. *Moon* signs an agreement with a customer, called *Blue*, to exchange purchase order messages in RosettaNet PIP 3A4 format. Currently, the back-end systems of *Moon* use proprietary data models and interaction protocols that differ from those of RosettaNet. The objective is to build a *Mediator* – a system that compensates the differences and enables *Blue* and *Moon* to interoperate (cf. Figure 7-116).

[34] http://www.w3.org/2005/Incubator/swsc/

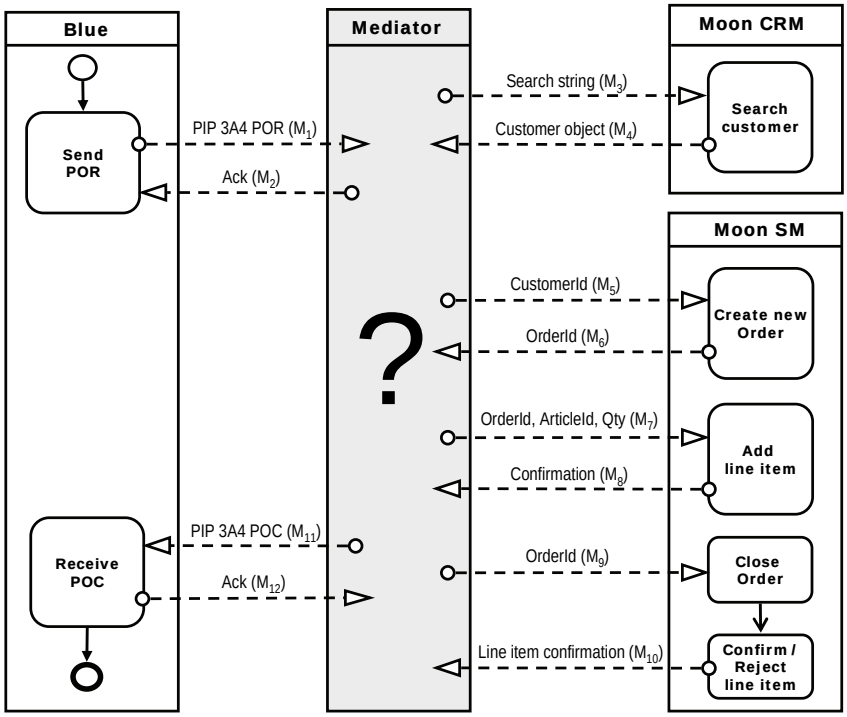


Figure 7-116 *Blue*'s and *Moon*'s systems

The interaction between systems is initiated by customer *Blue* who sends a *PIP 3A4 Purchase Order Request* (message M_1)[35]. *RosettaNet PIP 3A4* enables a buyer to issue a purchase order and to obtain a quick response (message M_{11}) from a seller acknowledging which of the purchase order product line items are *accepted*, *rejected*, or *pending*. Both messages M_1 and M_{11} must be synchronously confirmed by an *Acknowledgement of Receipt (*messages M_2 and M_{12}, respectively).

RosettaNet messages do not contain specific information about products, but only *global unique product identifiers*. For that purpose RosettaNet uses the *Global Trade Identification Number* (GTIN). GTIN is the EAN.UCC System identifier for trade items, which defines both products and services. GTINs provide the capability to deliver unique identification worldwide. An example of GTINs is given in *Table 7-1*.

[35] The XML schemata of all messages are presented in Appendix B.

Table 7-1
Example of GTINs

Description	Item	Packaging	GTIN
Dell W5001C 50" High Definition Plasma TV	*1 Unit*	*Consumer*	*00614141000012*
SANDISK 1 GB Secure Digital Card	*96 Units*	*Case*	*00614141000029*
Dell Laser 1710	*6 Pack*	*Consumer*	*00614141000883*
DELL 512 MB High Speed USB 2.0 Memory Key	*8x12 Pack*	*Case*	*50614141000994*

Blue sends a *Purchase Order Request* message containing customer information and multiple line items. In order for *Moon* to process this message, several steps have to be performed.

First, a *Search string* has to be sent to the *CRM* system (message M_3) to check whether a customer is known to *Moon*. If this is the case, the *CRM* system will reply with a message (M_4) containing a *Customer object* that matches the search string. Next, a message (M_5) has to be sent to the *SM* system to *create a new order* to which the *SM* system replies synchronously with a message (M_6) containing the *OrderId* of the newly created order. Once a new order is created, all line items have to be *added* one by one to that order by sending multiple messages (M_7). These messages are *confirmed* synchronously by the *SM* system (message M_8). After all line items are added to the order, a message (M_9) has to be sent to the *SM* system to *close the order*. Upon receiving this message the *SM* system starts sending multiple messages (M_{10}) to *confirm the status* of each line item. Once all order lines are processed by *Moon*, a *RosettaNet PIP3A4 Purchase Order Confirmation* message (M_{11}) has to be sent to the customer *Blue* and be confirmed synchronously by an *Acknowledgement of Receipt* message (M_{12}).

In the remainder of this section, we apply each step of our integration method to solve the integration problem described above. To do this, we apply the steps of our integration method using *concrete technologies*, i.e., WSDL and WS-BPEL (for modeling the service PSMs) and COSMO in conjunction with OWL-DL (for modeling the service PIMs).

First, we start with the WSDL descriptions of the *Blue* and *Moon* services and derive their service PIMs. Note that the information within a WSDL description only defines the messages accepted by a system, but does not define the messages sent by that system. In addition, a WSDL description does not define the order of the exchanged messages (i.e., the interaction protocol of the system). Therefore, to derive the complete PIMs of *Blue* and *Moon* we use the provided textual description of the integration problem and consult the RosettaNet specification. Once, we have derived the service

PIMs of *Blue* and *Moon*, we *enrich* these models with additional semantics, making implicit knowledge about the systems more explicit. Next, we solve the integration problem by specifying the information and behavior PIMs of the *Mediator*. The PIM allows the same solution to be implemented using different software technologies (i.e., the same abstract solution is reused to implement different concrete solutions). Once we have specified the service PIM of the *Mediator*, we check whether the integrated systems (consisting of *Blue*, *Moon* and the *Mediator*) meets the necessary conditions for semantic and pragmatic interoperability (as defined in Chapter 5). Finally, we automatically derive the service PSM of the *Mediator* by transforming its service PIM to executable specification (in terms of WS-BPEL an additional data transformation web service).

7.2.1 Step 1. Abstracting WSDL descriptions of Blue and Moon to COSMO

In Step 1 (cf. Figure 7-117), we perform two activities, namely, (i) we derive the *information* PIMs and (ii) the *behavior* PIMs of *Blue* and *Moon* using the WSDL descriptions of their services.

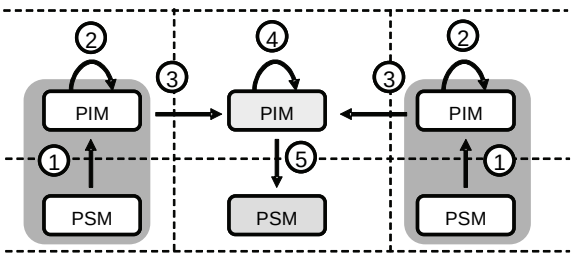


Figure 7-117 Step 1. Abstracting WSDL descriptions of *Blue* and *Moon* to COSMO

Deriving the Information PIMs

We derive the information PIMs of *Blue* and *Moon* from the *types* sections of the WSDL descriptions of their services. We do this by adopting and extending the rules defined in (Battle, 2006; García, 2005). The mapping rules are summarised in the table below:

XML Schema	*OWL*
1. attribute	*DatatypeProperty*
2. element of a simple type	*DatatypeProperty*
3. element of a complex type	*ObjectProperty*
4. simple type	*DatatypeProperty*
5. complex type	*Class*
a. complex type (sequence or all)	*intersectionOf*
b. complex class (choice)	*intersectionOf(unionOf, complementOf)*
6. restriction / extension of a complex type	*subClassOf*
7. restriction on a simple type	*subPropertyOf*
8. minOccur / maxOccur constraint	*minCardinality / maxCardinality restriction*

In the following, we present each rule and give a short example to illustrate it. All rules are generic; they can be applied to transform any XML schema to an OWL ontology.

1. An XML schema *attribute* is transformed to an OWL *DatatypeProperty*.

Example

```
<xsd:schema ...>
 ...
 <xsd:attribute name="name" type="xsd:string"/>
 ...
</xsd:schema>
```

```
:name a owl:DatatypeProperty;
  owl:dataRange xsd:string .
```

2. An XML schema *element* of a complex type is transformed to an OWL *ObjectProperty*.

Example

```
<xsd:schema ...>                                  complex type
 ...                                                  ↓
 <xsd:element name="address" type="AddressType"/>
 ...
</xsd:schema>
```

```
:address a owl:ObjectProperty;
  rdfs:range :AddressType .
```

3. An XML schema *element* of a simple type is transformed to an OWL *DatatypeProperty*.

Example

```
<xsd:schema ...>
 ...
 <xsd:element name="city" type="xsd:string"/>
 ...
</xsd:schema>
```

```
:city a owl:DatatypeProperty;
  rdfs:dataRange xsd:string .
```

4. An XML schema *simpleType* is transformed to an OWL *DatatypeProperty*.

Example

```
<xsd:schema ...>
 ...
 <xsd:simpleType name="adultAge">
   <xsd:restriction base="integer">
     <xsd:minInclusive value="18">
   </xsd:restriction>
 </xsd:simpleType>
 ...
</xsd:schema>
```

```
:adultAge a owl:DatatypeProperty;
  rdfs:domain rdfs:subClassOf
    [ a owl:DatatypeRestriction;
      owl:onDataRange xsd:nonNegativeInteger;
      owl:minInclusive "18"^^xsd:integer ].
```

5. An XML schema *complexType* is transformed to an OWL *Class*. More precisely, (a) a *complexType* using the compositors *sequence* or *all* is transformed to an OWL *Class* defined as *intersectionOf* of property restrictions derived from the elements of the compositor.

Example

```
<xsd:schema ...>
 ...
 <xsd:complexType name="Address">
   <xsd:sequence>
     <xsd:element name="street" type="xsd:string"/>
     <xsd:element name="city" type="xsd:string"/>
   </xsd:sequence>
 </xsd:complexType>
 ...
</xsd:schema>
```

```
:Address a owl:Class;
  owl:equivalentClass
  [ owl:intersectionOf (
    [ a owl:Restriction; owl:onProperty :street;
      owl:allValuesFrom xsd:string ]
    [ a owl:Restriction; owl:onProperty :city;
      owl:allValuesFrom xsd:string ]
 )].
```

(b) A *complexType* using the compositor *choice* is transformed to an OWL *Class* defined as an expression containing OWL *intersectionOf, unionOf* and *complementOf*. If fact, the *choice* (i.e., exclusive *or*) is defined as combination of conjunctions (*and*), disjunctions (*or*), and negations (*not*), e.g. the *choice* between *A and B* is equivalent to *(A or B) and not (A and B)*.

Example

```
<xsd:schema ...>
  ...
  <xsd:complexType name="ShippingAddressType">
    <xsd:choice>
      <xsd:element name="dutchAddress" type="DutchAddressType"
      <xsd:element name="ukAddress" type="UKAddressType"
    </xsd:choice>
  </xsd:complexType>
  ...
</xsd:schema>
```

```
:ShippingAddressType a owl:Class;
  owl:intersectionOf (
    owl:unionOf (
      [ a owl:Restriction; owl:onProperty :dutchAddress;
        owl:allValuesFrom :DutchAddressType ]
      [ a owl:Restriction; owl:onProperty :ukAddress;
        owl:allValuesFrom :UKAddressType ] )
    owl:complementOf owl:intersectionOf (
      [ a owl:Restriction; owl:onProperty :dutchAddress;
        owl:allValuesFrom :DutchAddressType ]
      [ a owl:Restriction; owl:onProperty :ukAddress;
        owl:allValuesFrom :UKAddressType ] )
   ) .
```

6. An XML schema *complexType* derived by *extension* from or *restriction on* another *complexType* is transformed to a *rdfs:subClassOf* assertion between the respective OWL classes.

Example

```
<xsd:schema ...>
  ...
  <complexType name="Car">
    <extension base="Vehicle">
  </complexType>
  ...
</xsd:schema>
```

```
:Car a owl:Class;
  rdfs:subClassOf :Vehicle.
```

7. An XML schema *simpleType* derived by *restriction* from another *simpleType* is transformed to a *rdfs:subPropertyOf* assertion between the respective OWL properties.

Example

```
<xsd:schema ...>
 ...
 <simpleType name="adultAge">
   <restriction base="age">
 </complexType>
...
</xsd:schema>
```

```
:adultAge a rdf:Property;
  rdfs:subPropertyOf :age.
```

8. XSD *minOccur* and *maxOccur* constraints are transformed to OWL *minCardinality* and *maxCardinality* property restrictions.

Example

```
<xsd:schema ...>
 ...
 <xsd:complexType name="Address">
   <xsd:sequence>
     <xsd:element name="street" type="xsd:string" minOccur="1"/>
     <xsd:element name="city" type="xsd:string" minOccur="1" />
   </xsd:sequence>
 </xsd:complexType>
 ...
</xsd:schema>
```

```
:Address a owl:Class;
  owl:equivalentClass
  [ owl:intersectionOf (
    [ a owl:Restriction; owl:onProperty :street;
      owl:allValuesFrom xsd:string ]
    [ a owl:Restriction; owl:onProperty :street;
      owl:minCardinality 1 ]
    [ a owl:Restriction; owl:onProperty :city;
      owl:allValuesFrom xsd:string ]
    [ a owl:Restriction; owl:onProperty :city;
      owl:minCardinality 1 ]
 )].
```

We apply the transformation rules on the *types* sections from the WSDL descriptions of *Blue* and *Moon,* and derive their information models in terms of OWL-DL. The complete information models are quite verbose, therefore we only present small excerpt of them in Turtle syntax (Beckett, 2007):

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix : <http://sws-challenge.org/schemas/rnet2#> .

 :AddLineItemResponseType a owl:Class;
   rdfs:subClassOf
      [ a owl:Restriction;
        owl:onProperty :lineItemId;
        owl:cardinality 1 ];
   rdfs:subClassOf
      [ a owl:Restriction;
        owl:onProperty :orderId;
        owl:cardinality 1 ];
   rdfs:subClassOf
      [ a owl:Restriction;
        owl:onProperty :lineItemId;
        owl:allValuesFrom xsd:long ];
   rdfs:subClassOf
      [ a owl:Restriction;
        owl:onProperty :orderId;
        owl:allValuesFrom xsd:long ];

:AddLineItemResponse rdfs:range :AddLineItemResponseType .

:Item a owl:Class;
   rdfs:subClassOf
```

```
    [ a owl:Restriction;
      owl:onProperty :articleId;
      owl:cardinality 1 ];
  rdfs:subClassOf
    [ a owl:Restriction;
      owl:onProperty :quantity;
      owl:allValuesFrom xsd:int ];
  rdfs:subClassOf
    [ a owl:Restriction;
      owl:onProperty :quantity;
      owl:cardinality 1 ];
  rdfs:subClassOf
    [ a owl:Restriction;
      owl:onProperty :articleId;
      owl:allValuesFrom xsd:string ] .
```

Deriving the Behavior PIMs

The behavior PIMs are derived using the *interface* section of the WSDL descriptions of *Blue* and *Moon*.

As presented in Chapter 3, a WSDL description contains two parts. In the abstract part, WSDL defines a web service in terms of messages accepted by the system that implements the service. The messages are defined by means of a type system (typically XML Schema) and their sequence and cardinality is defined by *message exchange patterns* (*MEPs*). An *operation* associates message exchange patterns with one or more messages. An *interface* is used to group these operations. In the concrete part of the WSDL description, bindings specify the transport and wire format for interfaces. A *service endpoint* associates network address with a binding. Finally, a *service* groups the endpoints that implement a common interface.

Using the WSDL descriptions of *Blue* and *Moon* we derive the initial behavior models of the systems (cf. Figure 7-118).

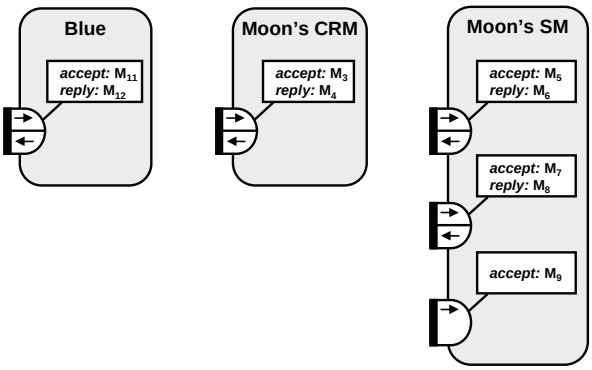


Figure 7-118
The behavior PIMs of *Blue* and *Moon*

M_1-M_{16} are the messages described in the beginning of this section. They are summarised in the following table:

M1	Purchase Order Request
M2	Acknowledgement of Receipt
M3	Search Customer Request
M4	Search Customer Response
M5	Create New Order Request
M6	New Order Response
M7	Add Line Item Request
M8	Add Line Item Response
M9	Close Order
M10	Order Line Item Confirmation
M11	Purchase Order Confirmation
M12	Acknowledgement of Receipt message

In this step, we provided evidence for the validity of Claim 1, i.e., *service PIMs can be derived from their respective PSMs.*

7.2.2 Step 2. Semantic Enrichment of the Service PIMs

In Step 2 (cf. Figure 7-119), we semantically enrich (i) the *information* PIMs and (ii) the *behavior* PIMs of *Moon* and *Blue*.

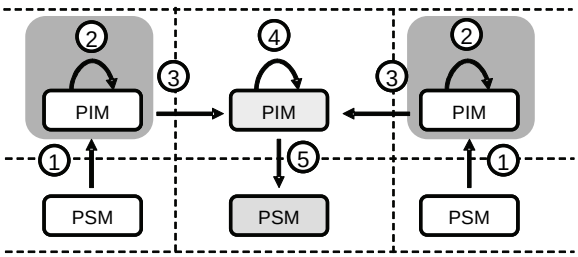


Figure 7-119
Step 2. Semantic enrichment of the service PIMs

Semantic Enrichment of the Information PIMs

A WSDL types section defines only the *syntax* of the messages to be exchanged between the provider of the service and its requestors. Further work is required to define the *semantics* of these messages. For example, all hidden assumptions should be made explicit by defining new classes and relations among them, or by mapping the classes and properties to classes and properties from some domain-specific ontologies and thus defining their meaning. This is usually a manual process which requires domain specific knowledge.

To illustrate one way to semantically enrich the information models of *Blue* and *Moon* we use the *Universal Data Element Framework* (UDEF, 2008). UDEF is an Open Group standard, which enables organizations to give

meaning to elements of the information models of their systems by tagging them with globally standard identifiers. These identifiers are constructed by concatenating unique identifiers of object classes and properties defined in the UDEF standard.

The UDEF trees are available as an OWL ontology. We link the elements of the information models of *Blue* and *Moon* systems to elements of UDEF ontology by defining *owl:equivalentProperty* or *rdfs:subPropertyOf*, relations among corresponding properties.

To illustrate the use of UDEF we present an excerpt of the semantically enriched information models of *Blue* and *Moon* (cf. Figure 7-120 and Figure 7-121).

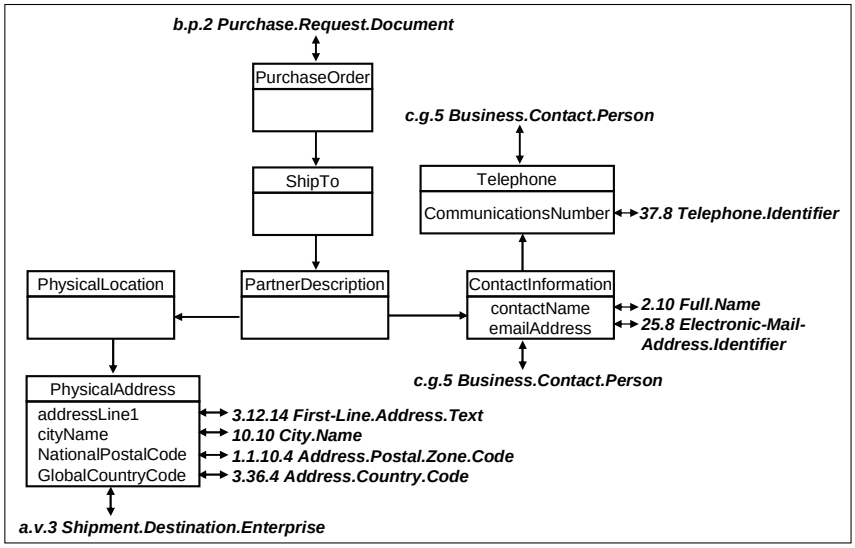


Figure 7-120
The semantically enriched PurchaseOrder (*Blue* system)

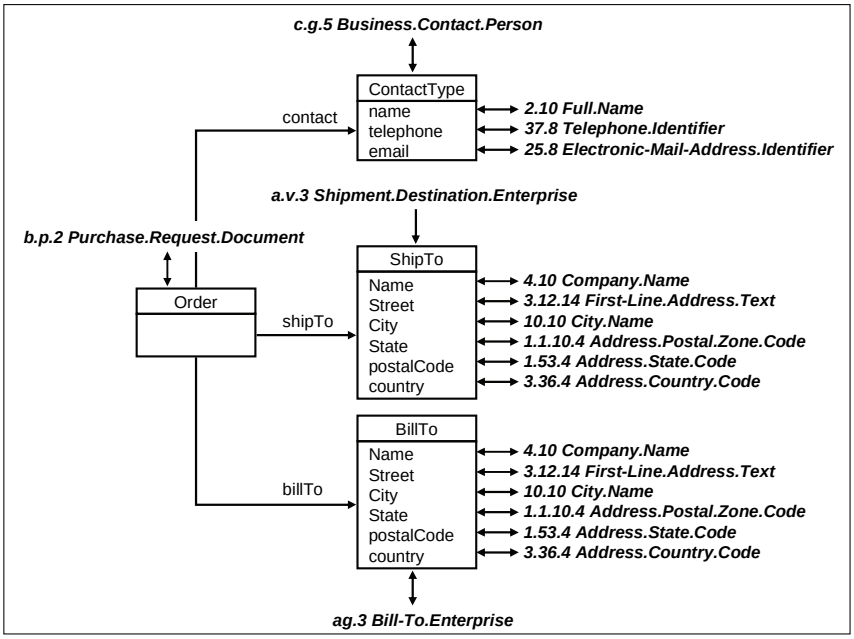


Figure 7-121
The semantically enriched Order (*Moon* system)

The semantic annotations are used in Step 3 to match automatically equivalent properties and to suggest mapping relations.

Semantic Enrichment of the Behavior models

A WSDL description only defines the operations, provided by some service, i.e., it only defines the *operation execution* part of an operation. To define an operation completely we need also to define the *operation call* part. Since this information is missing in the WSDL descriptions of *Blue* and *Moon*, we use the textual description of the integration problem as well as the abstract diagram of the *Mediator*. The behavior models of *Blue* and *Moon*, after adding the operations' calls, is presented in Figure 7-122.

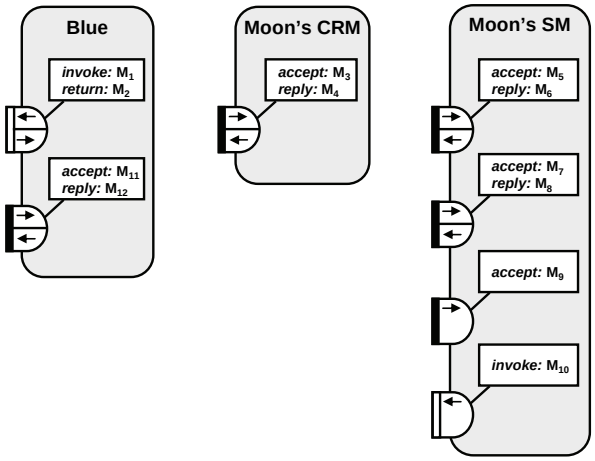


Figure 7-122
The semantically enriched behavior PIMs of *Blue* and *Moon*

A WSDL *message exchange pattern (MEP)* defines only the relationship between (input, output and fault) messages of a *single* operation. The complete behavior model should also define the relationships between the different operations. Since these relationships are not part of the WSDL descriptions, they have to be derived from the informal textual descriptions as provided in scenario description. In addition, the repetitive steps should be made explicit as well. The semantically enriched behavior models of *Blue* and *Moon* are presented in Figure 7-123.

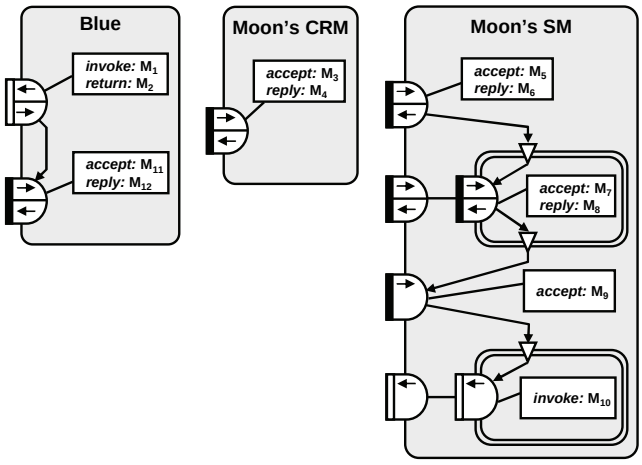


Figure 7-123
The semantically enriched behavior PIMs of *Blue* and *Moon*

In this step, we provided evidence for the validity of Claim 3, i.e., *the service models of the systems to be integrated can be semantically enriched.*

7.2.3 Step 3. Solving Integration Problem at PIM Level

In Step 3 (cf. Figure 7-124), we *design* the information and behavior models of the *Mediator*.

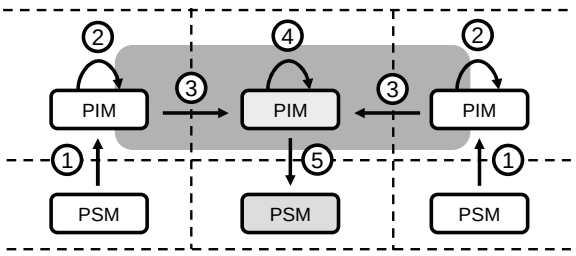


Figure 7-124 Step 3. Solving integration problem at PIM level

The information model of the *Mediator* is constructed from the union of the information models of *Blue* and *Moon*. In addition, the information model of the *Mediator* contains new classes to represent status information of the *Mediator* (e.g., the set of order line items that have been confirmed so far) as well as the mapping relations among the classes and properties from the information models of *Blue* and *Moon*.

The construction of the behavior model of the *Mediator* requires the definition of (i) the mapping relations among the classes and properties from the information models of *Blue* and *Moon*, (ii) the services provided and requested by the *Mediator* and (iii) the composition of these services by relating their operations.

To define the mappings among the classes and properties from the information models of *Blue* and *Moon* we use the approach presented in (Haase, 2005).

An OWL mapping system `MS` is a triple (`S, T, M`), where `S` is the source information model, `T` is the target information model, and `M` is the mapping between `S` and `T`, i.e., a set of assertions $Qs \rightarrow Qt$, where `Qs` and `Qt` are conjunctive queries over `S` and `T`, respectively, with the same set of distinguished variables `x`. Thus, a mapping is equivalent to an axiom:

$$\forall x: Qs(x, y_s) \rightarrow Qt(x, y_t)$$

The correspondence between classes and properties from the information models of *Blue* and *Moon* are expressed as a function of subsumption, e.g., $Qs \subseteq Qt$. Using an OWL reasoner, such as Racer[36]and Pellet[37], allows us to check the information model of the *Mediator* for consistency.

[36] Racer Systems, Racer Reasoner, http://www.racer-systems.com/

To facilitate the use of the mapping approach described above, we have defined a *Domain-Specific Mapping Language* (*mapping DSL*). The mapping DSL provides a means for defining the mapping relations between the information models of *Blue* and *Moon* and for deriving automatically the information model of the *Mediator* in OWL (used in the verification step to check necessary condition 1) and the Java classes that implement the data transformation (used at runtime to transform the exchanged messages between *Blue* and *Moon*). The metamodel of the mapping DSL is presented in Figure 7-125.

A *Transformation* consists of one or more *Mapping*s. A *Mapping* defines an assertion `Qs` → `Qt` where *Qs* is defined as conjunction of `Bindings` in a number of *Source domains* and *Qt* is defined in a Binding in the *Target* domain. In addition, a *Mapping* may contain zero or more expressions which bind variables by invoking other Mappings or custom functions.

Figure 7-125
The metamodel of the mapping DSL

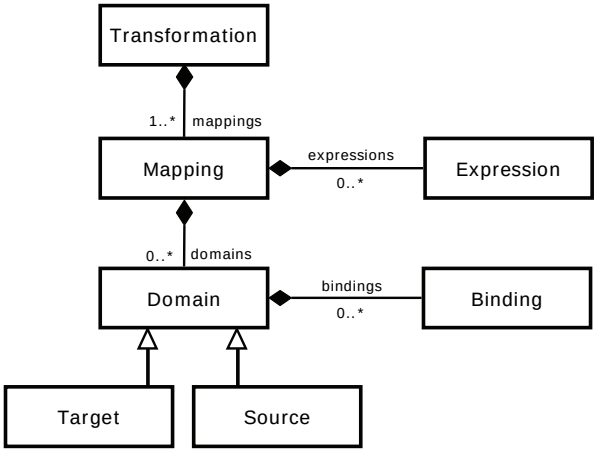


Figure 7-126 illustrates the mappings between the class *Pip3A4PurchaseOrderRequest* from the information model of *Blue* and the classes *SearchCustomer*, *Order* and *LineItem* from the information model of *Moon*. Note, that the mappings are discovered automatically using the UDEF information added in Step 2.

[37] http://clarkparsia.com/pellet/

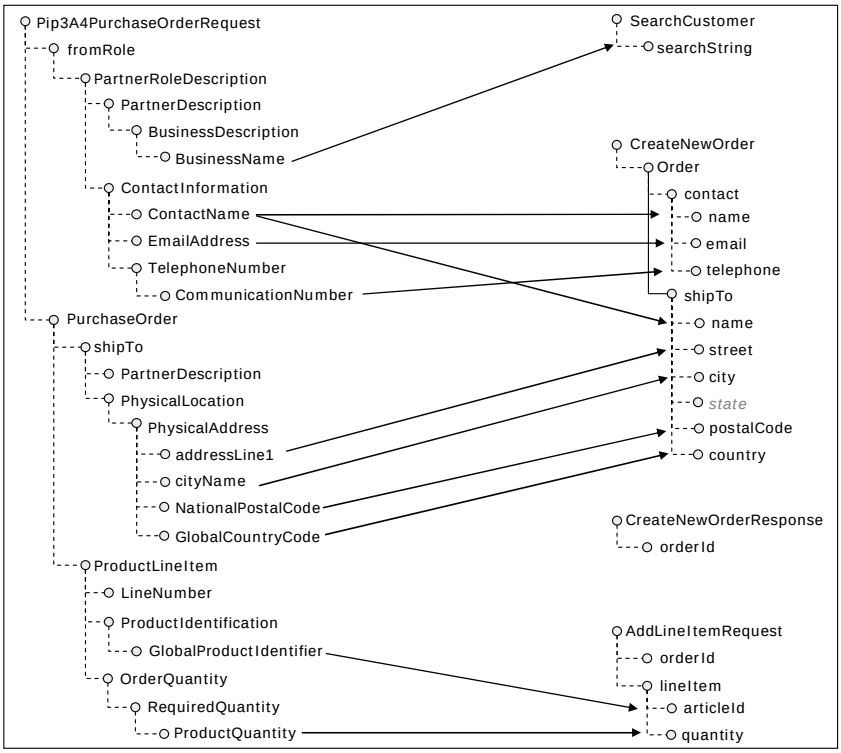


Figure 7-126
The mapping between the information PIMs of *Blue* and *Moon*

Using the mapping DSL we formally define these mappings:

```
transformation Blue2Moon {
  mapping POR2Search {
    source por:Pip3A4PurchaseOrderRequest {
      fromRole(?por, ?role)
      PartnerRoleDescription(?role, ?partnerRole)
      PartnerDescription(?partnerRole, ?partner)
      BusinessDescription(?partner, ?business)
      BusinessName(?business, ?businesName)
    }
    target search:SearchCustomer {
      searchString(?search, ?businesName)
    }
  }

  mapping POR2NewOrder {
    Source por:Pip3A4PurchaseOrderRequest {
      fromRole(?por, ?role)
        PartnerRoleDescription(?role, ?partnerRole)
          ContactInformation(?partnerRole, ?contact)
            ContactName(?contact, ?contactName)
            EmailAddress(?contact, ?email)
            TelephoneNumber(?contact, ?telephone)
            CommunicationNumber(?telephone, ?phoneNumber)
            purchaseOrder(?por, ?order)
         shipTo(?order, ?shipTo)
```

```
            PhysicalLocation(?shipTo, ?location)
              PhysicalAddress(?location, ?address)
                addressLine1(?address, ?street)
                cityName(?address, ?city)
                NationalPostalCode(?address, ?pcode)
                GlobalCountryCode(?address, ?country)
    }
    target order: Order {
      contact(?order, ?newContact)
        name(?newContact, ?contactName)
        email(?newContact, ?email)
        telephone(?newContact, ?phoneNumber)
      shipTo(?order, ?newShipTo)
        name(?newShipTo, ?contactName)
        street(?newShipTo, ?street)
        city(?newShipTo, ?city)
        postalCode(?newShipTo, ?pcode)
        country(?newShipTo, ?country)
    }
  }

  mapping POR2NewLineItem {
    Source por:Pip3A4PurchaseOrderRequest {
      purchaseOrder(?por, ?order)
        ProductLineItem(?order, ?lineItem)
          ProductIdentification(?lineItem, ?id)
            GlobalProductIdentifier(?id, ?globalProductId)
      OrderQuantity(?order,?orderQty)
        RequiredQuantity(?orderQty, ?requredQty)
          ProductQuantity(?requredQty, ?quantity)
    }
    Target item: lineItem {
      articleId(?item, ?globalProductId)
      quantity(?item, ?quantity)
    }
  }
}
```

The *Mediator* provides one service that must match the service requested by *Blue*. The service provided by the *Mediator* can initially be defined by "mirroring" the service requested by *Blue*. The mirroring of a service is obtained by changing each *operation call* into an *operation execution*, and vice versa, while keeping the same parameters. In addition, the relations among the operations and the parameter constraints may (initially) be retained. Likewise, the services that are requested by the *Mediator* can be obtained by mirroring the services that are provided by *Moon*.

The design of the *Mediator* behavior can now be approached as the search for a composition of the requested services that conforms to the provided service. The structure of this composition is defined by the (causal) relations among the operations. We do this by inspecting the mapping relations in the information model of the *Mediator*. For instance, the elements of M_3 are related to the elements of M_2 by mapping relation *POR2Search*, i.e., the information required to construct M_3 is provided in

M_1. Therefore, we use the *message splitting* pattern (see Chapter 5, p.23) . Similarly, the elements of M_5 are related to the ones of M_1 by mapping relation *POR2NewOrder*. In addition, M_5 contains the *orderId* (provided by M_4) i.e., the information required to construct M_5 is provided in messages M_1 and M_4. Therefore, we instantiate the *message aggregation* pattern (see Chapter 5, p.23).

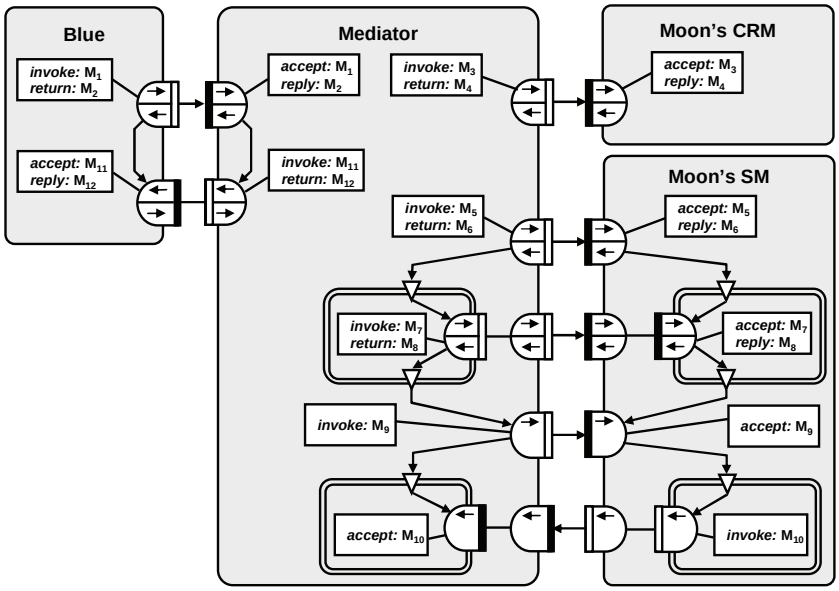


Figure 7-127
The behavior model of the *Mediator* after mirroring operation calls and operation executions

The information mappings are not sufficient to define the complete behavior of the *Mediator*. Although the search for a customer (M_3) in CRM system gives information (M_4) such as *AddressInfo* or *ContactInfo*, still the information that is provided in M_1 should be used instead. Such hidden assumptions have to be made explicit in the behavior model of the *Mediator*. The complete behavior model of the *Mediator* is presented in Figure 7-128.

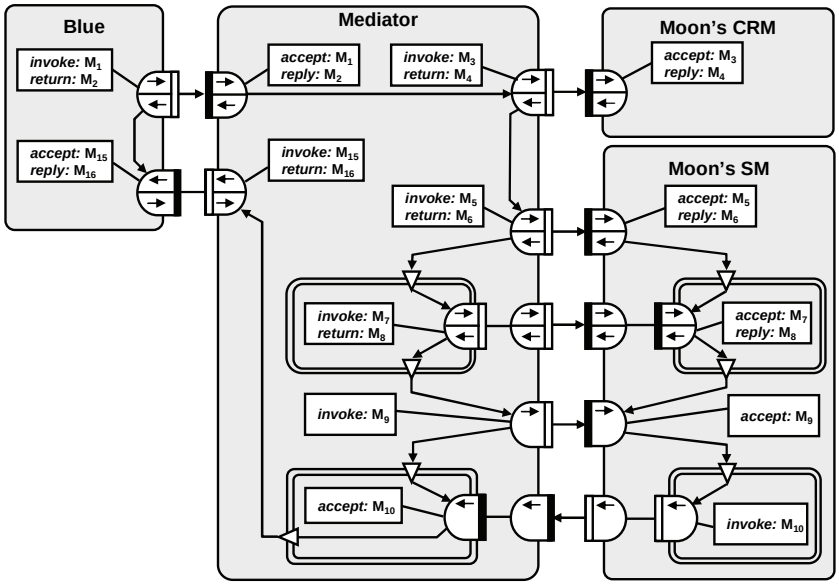


Figure 7-128
The complete behavioral model of the mediator

In this step, we provided evidence for the validity of Claim 2, i.e., *COSMO provides all required concepts to model platform-independent integration solutions*.

7.2.4 Step 4. Verification of the integration solution

In Step 4 (cf. Figure 7-129) of our integration method, we analyse whether or not the proposed integration solution enables the integrated systems to interoperate.

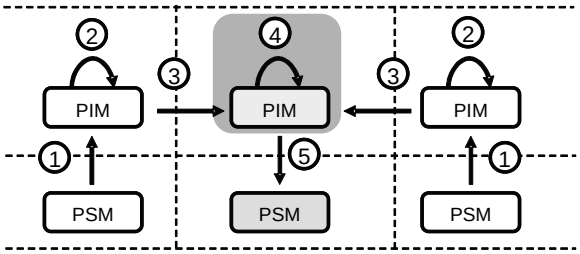


Figure 7-129
Step 4. Verification of the integration solution

To do this, we first abstract from the participation of each system and construct the behavior of the integrated system. Next, we transform the integrated behavior to a Petri Net and construct the corresponding occurrence graph. Finally, we use the occurrence graph to check whether the integrated systems are pragmatically interoperable (see Chapter 5, Section 5.1.3)

The integrated behavior of *Blue*, *Moon* and the *Mediator* systems after representing the concurrent executions, choices, conjunctions and disjunctions of conditions explicitly is shown in Figure 7-130.

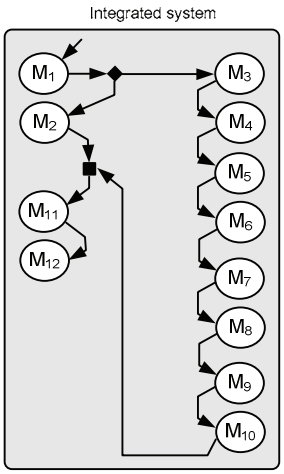


Figure 7-130
The integrated perspective of the *Blue*, *Moon* and the *Mediator*

After applying the transformation rules described in Appendix A, we derive the corresponding Petri Net (Figure 7-131).

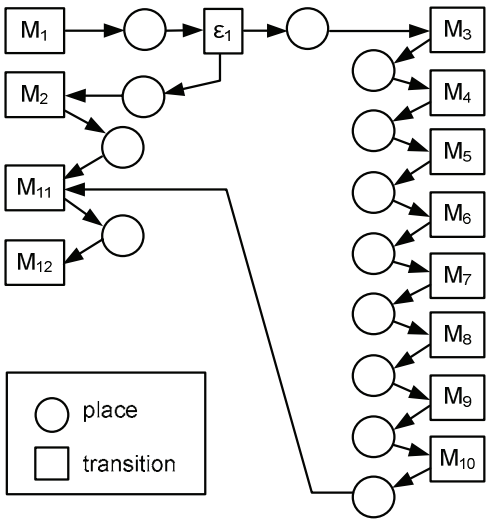


Figure 7-131
The Petri Net of the integrated system

Next, we construct the occurrence graph of the net (Figure 7-132)

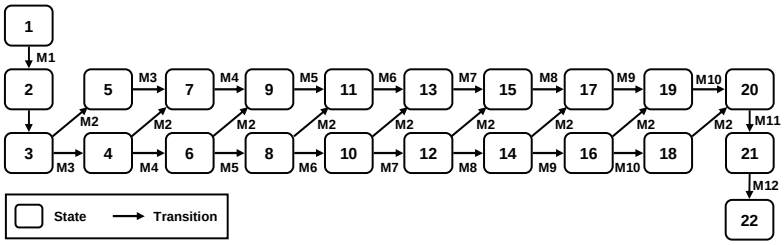


Figure 7-132 The occurrence graph of the Net in Figure 7-131

Finally, we analyse the occurrence graph to check whether the integrated system supports the desired execution. In our case, we check whether M_{12} can be sent after M_1 has been sent (one possible execution trace is depicted in Figure 7-133). Based on the result of the query on the occurrence graph, we can conclude that systems satisfy the third necessary condition for interoperability (see Chapter 5).

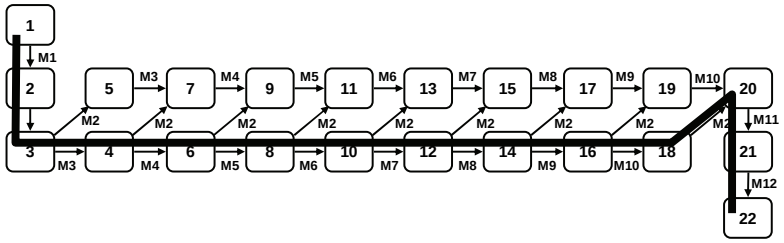


Figure 7-133 Checking whether M_{16} is reachable from M_1 via M_{11} and M_{12}

7.2.5 Step 5. Deriving the PSM of the integration solution

In Step 5 (cf. Figure 7-134), the service PIM of the *Mediator* is transformed into a platform-specific model in terms of WS-BPEL.

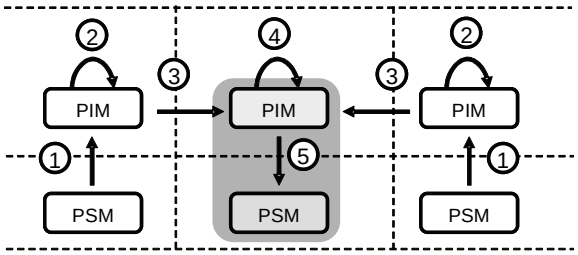


Figure 7-134 Step 5. Deriving the PSM of the integration solution

In Chapter 5, we presented the abstract architecture of the *Mediator*. To recap, it consists of two main components: a *Control Flow Manager* and a *Data Flow Manager*. The *Control Flow Manager* is responsible for sending and receiving messages in a particular order as well as for querying and updating the state of the *Mediator*. The *Data Flow Manager* in turn, is responsible for

managing the state of the *Mediator* and for performing the necessary data transformations and constraint checking.

For this case, we have selected WS-BPEL engine to implement the *Control Flow Manager*. Therefore, the behavior PIM of the *Mediator* has been mapped to a WS-BPEL specification. The *Data Flow Manager* has been implemented as separate Web Service. The reason for that is to provide an interpreter for the mapping DSL and execute the data transformations at run-time.

To derive the *Control Flow Manager* we adopt and extend the approach described in (Dirgahayu, 2007). In this approach, the transformation is divided into three successive transformation steps: (T_1) *pattern recognition*, (T_2) *activity replacement,* and (T_3) *model realization* (see Figure 7-135).

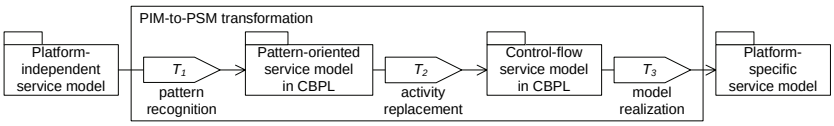


Figure 7-135 Transforming the service PIM of the *Mediator* to a service PSM

In the first step (T_1), control flow patterns from service the PIM of the *Mediator* are recognised and then transformed to the pattern-oriented service model in terms of *Common Behavioral Patterns Language* (CBPL). Each CBPL pattern represents a control flow that is common to most execution languages, i.e., *sequence*, *concurrence*, *selection,* and *iteration (cf.* Figure 7-136*)*. A *sequence* contains one or more activities to be executed in succession. A *concurrence* contains two or more activities that can be executed independently, i.e., in parallel. *Iteration* contains one or more activities to be executed repeatedly as long as a condition holds. *Selection* contains one or more *cases* to be selected. *Case* contains an activity to be executed when its condition holds.

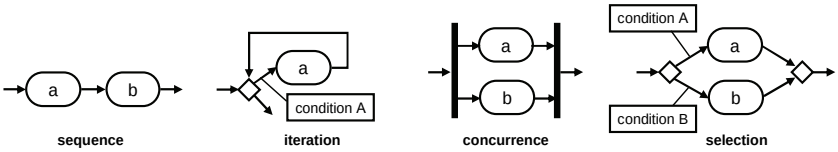


Figure 7-136 CBPL patterns: Sequence, concurrence, selection and iteration

In the second step (T_2), all data transformations and constraint evaluation activities from the pattern-oriented service model are replaced with operations to interact with the *Data Flow Manager*. This step results in a control-flow service model that represents the *Control Flow Manager* in CBPL.

In the last step (T_3), the control-flow service model is mapped onto a service PSM in terms of WS-BPEL. Note, that a service PSM contains

information that is not present in the service PIM. Examples of such information are the XML namespaces of the exchanged messages or the WSDL port types and operations of the services to be integrated. To provide the required platform-specific information we annotate the elements of the service PIM. This information is maintained during the first and second step and is used in the third step. In the following, we present the mapping rules from a service model in CBPL to a WS-BPEL description.

Sequence maps to *bpel:sequence*

```
<sequence>
   <a />
   <b />
</sequence>
```

Iteration maps to *bpel:while*

```
<while>
   <condition>
      Condition A
   </condition>
   <a />
</while>
```

Concurrence maps to *bpel:flow*

```
<flow>
   <a />
   <b />
</flow>
```

selection maps to *bpel:if*

```
<if>
  <condition>
      Condition A
  </condition>
  <a />
  <elseif>
    <condition>
       Condition B
    </condition>
    <b />
  </elseif>
</if>
```

In some cases, the behavior of the mediator may need to be restructured to enable mapping onto CBPL patterns. Figure 7-137 gives examples of such a restructuring.

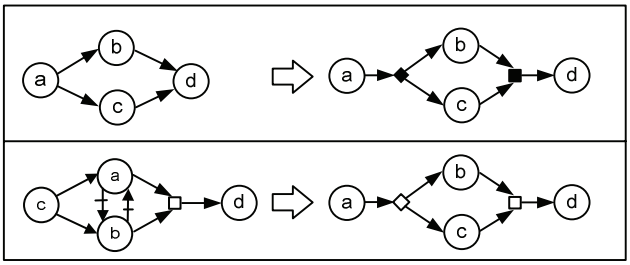


Figure 7-137 Behavior restructuring

The excerpt of the abstract WS-BPEL behavior of the *Mediator* is shown below

```
<sequence>
  <receive ...
    operation="receivePurchaseOrderRequest"
    inputVariable="M1"/>
  <invoke ...
    operation="updateState"
    inputVariable="M1"/>
  <flow>
    ...
    <sequence>
    <!-- create a variable to request M3, e.g.
      <requestM3>
        <mappingName>por2search</mappingName>
         <mappingParameter>M1</mappingParameter>
      </requestM3>
    -->
    <invoke ...
      operation="retrieveState"
      inputVariable="requestM3"
      outputVariable="M3"/>
    <invoke ...
      operation="search"
      inputVariable="M3"
      outputVariable="M4"/>
    <invoke ...
     operation="updateState"
     inputVariable="M4"/>
     ...
       <while>
      <invoke ...
        operation="addLineItem"
        inputVariable="M7"
        outputVariable="M8" />
      <invoke ...
        operation="updateState"
        inputVariable="M8"/>
     </while>
  ...
</sequence>
```

At run-time, the Web service, implementing the *Data Flow Manager,* receives data in XML format from the *Control Flow Manager* (operation *updateState*).

First, it transforms the data from XML to OWL and uses them to infer new state information, perform transformations and evaluate constraints. When requested (operation *retrieveState*), the Web service transforms back the data from OWL to XML and sends it to the *Control Flow Manager* for further use.

7.3 Scenario 2

To show that our integration method is able to cope with changes in the integration requirements, we present a second case defined in Scenario 2 of SWSC.

Moon decides to integrate also its *Production Management (PM)* system. The *Mediator* can use the *PM* to order products to be scheduled for production, when they are not available from *SM* system. (cf. Figure 7-138).

In addition to Scenario 1, if the *SM* system reports that an item is *not available,* the *PM* system will be used to check whether that item *can be produced.* This is done by sending a message (M_{13}) to the *PM* system to which this system responds synchronously by sending a message (M_{14}) containing the *price* and the *availability date*. If the *price* and the *availability date* meet the expectations of the customer *Blue* (as specified in message M_1) the item will be *ordered* by sending a message (M_{15}) to the *PM* system and be confirmed synchronously (M_{16}).

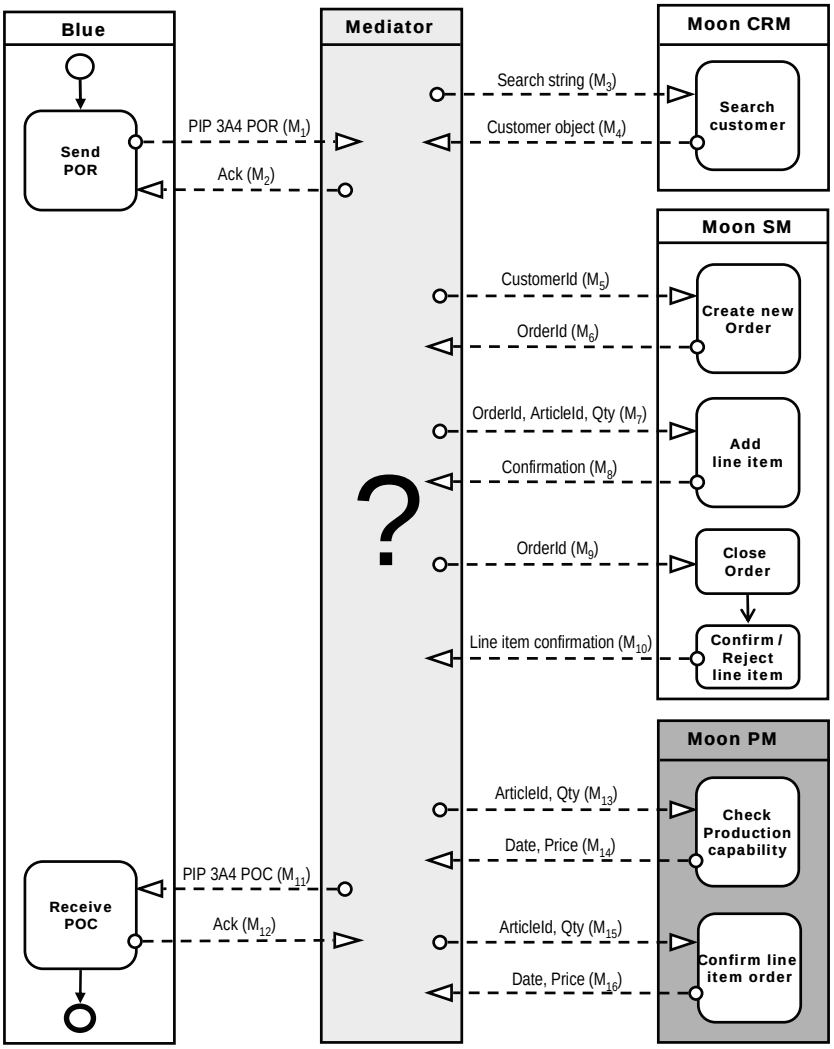


Figure 7-138
Blue's and *Moon*'s systems

In the remainder of this section, we apply each step of our integration method to solve the integration problem presented above. For that purpose, we apply the steps of our integration method again using *concrete technologies*, i.e., WSDL and WS-BPEL (for modeling the service PSMs) and COSMO in conjunction with OWL (for modeling the service PIMs).

7.3.1 Step 1. Abstracting WSDL description of PM to COSMO

In Step 1, we reuse the transformation defined in Section 7.2.1 and derive the information and behavior PIM of the PM system (cf. Figure 7-139)

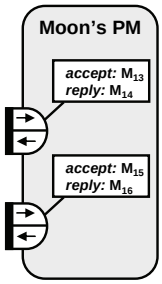


Figure 7-139
The PIM of the PM system

M_{13}-M_{16} are the messages described in the beginning of this section and summarised in the following table

M_{13}	Check Production Capability Request
M_{14}	Check Production Capability Response
M_{15}	Confirm Production Order Request
M_{16}	Confirm Production Order Response

7.3.2 Step 2. Semantic Enrichment of the service PIMs

This step is identical to Step 2 from Section 7.2.2, i.e., we use UDEF to semantically enrich the information model of the PM system. There is no need to semantically enrich the behavior PIM of the PM system.

7.3.3 Step 3. Solving integration problem at PIM level

In this step, we update the information and behavior models of the *Mediator* to reflect the changes in the business requirements. Similar to Step 3 from Section 7.2.3, we first identify and specify the new mapping relations between the information models of the *Mediator* and the *PM* system. Next, we obtain new services requested by the *Mediator* by mirroring the service provided by the *PM* system. Finally, we specify the conditions of the new services to address the new business requirements. The resulting behavior PIM of the *Mediator* is presented in Figure 7-140.

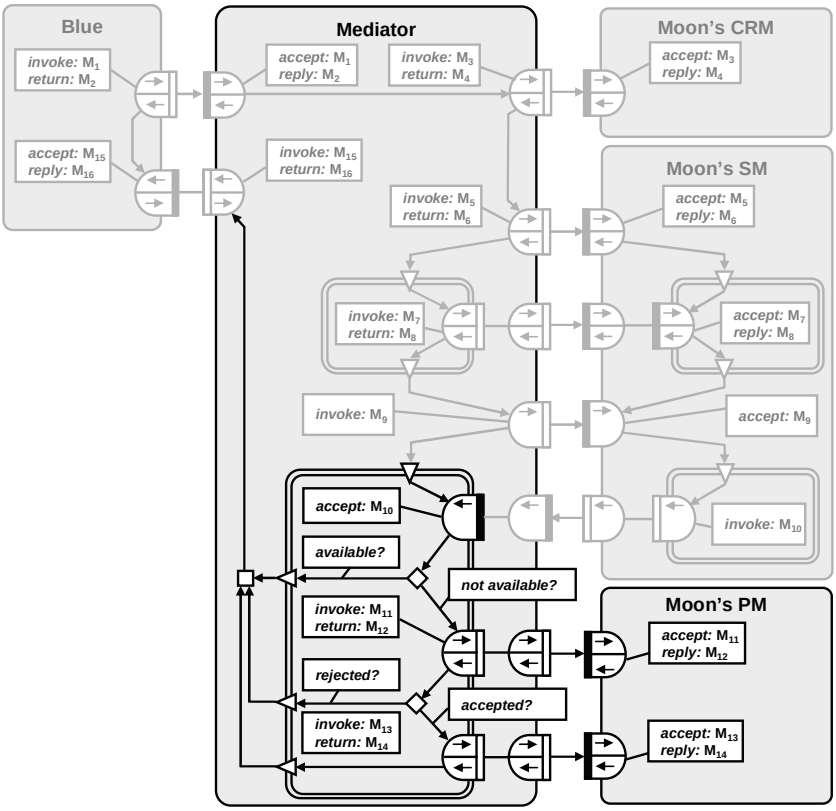


Figure 7-140
The complete behavioral model of the *Mediator*

7.3.4 Step 4. Verification of the integration solution

After updating the service PIM of the mediator to address the new requirements, we analyse whether or not the integration solution still enables the integrated systems to interoperate. Similar to Scenario 1, we first abstract from the participation of each system and construct the behavior of the integrated system. Next, we transform the integrated behavior to a Petri Net and construct its occurrence graph. Finally, we analyse this occurrence graph to check if the integrated system supports the desired execution. In our case, we check whether M_{16} can be send after M_1 is sent (cf. Figure 7-141). In addition, we can perform additional checks, for instance whether M_{16} is reachable from M_1 via M_{14} and M_{15} (cf. Figure 7-141). Based on the result of the queries on the occurrence graph, we can conclude that systems satisfy the third necessary condition for interoperability (see Chapter 5).

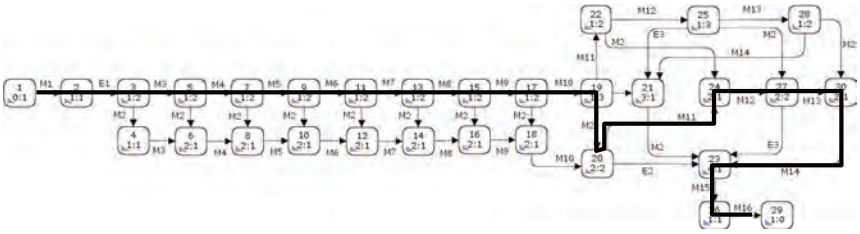


Figure 7-141 Checking whether M_{16} is reachable from M_1 via M_{14} and M_{15}

7.3.5 Step 5. Deriving the PSM of the integration solution

In this final step, the service PIM of the *Mediator* is transformed into a platform-specific model in terms of WS-BPEL. This is automatic step that reuses the transformation described in Step 5 of Section 7.2.5.

7.4 Summary

In this chapter, we applied our method for the semantic integration of service-oriented applications in a concrete context. That is, we solved the integration problems from Scenario 1 and Scenario 2 of SWSC using concrete SOA, MDA and KR technologies.

When applying the method we observed a number of effects. The observations supported the validity of the claims made in Chapter 6 and, in this way, it validated that the methods meets the requirements defined in Chapter 1.

In Chapter 9, we will compare the case, presented in this chapter, with a second one (presented in Chapter 8) and identify some commonalities and differences. By doing this we will seek to provide further insight into the applicability of our integration method in a more general context. In addition, we will analyse the observations that we made when solving the cases and will argue about to what extent our integration method meets the requirements defined in Chapter 1. Finally, we will present some lessons learnt.

Chapter 8

Railroad Operator Case

In this chapter, we validate our method in a second case. In this case, an existing integration solution is replaced with a new one in order to address new integration requirements.

The chapter is organised as follows: in Section 8.1, we present a real-world integration problem from the travel domain. In Section 8.2, we apply the steps of our integration method to solve the problem presented in Section 8.1. In Section 8.3, we show how our method addresses changes in the implementation technology. This is done by solving a problem presented in a hypothetical variation of the case from Section 8.1. Finally, in Section 8.4, we provide a short summary of the presented case.

8.1 Introduction

To validate our integration method in a second case study, we performed a lab experiment, i.e., we applied our method using real-world data. The data to perform the case study has been collected in the master thesis research of Bob Koehoorn (Koehorn, 2007). Bob worked for a company that was contracted to develop an integration solution for one of the major European railroad operators.

The IT architecture of the information systems for selling international train tickets in Europe is quite complicated. The major reason for this situation is that the physical railroad network is divided into various segments, managed by different railroad operators. Each of these operators uses its own proprietary information systems.

Overtime, railroad operators have built various information systems to handle different aspect of train travel (such as checking price and availability or booking a trip). To support the booking of international trips, some of these information systems have been integrated in order to provide two sales channels. The first one, called *@tlantis,* is used for selling tickets

through travel agencies. The second one, called *DirectMode*, is used for selling tickets by service desk employees[38]. Figure 8-142 depicts the current system architecture.

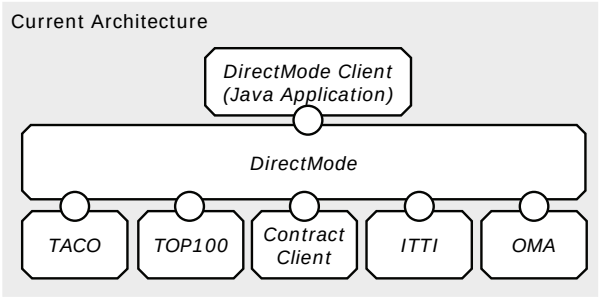


Figure 8-142
Current system architecture

TACO integrates and provides tariff and pricing information from various inventory systems used by the different railway operators. *TOP100* provides up-to-dates service schedules for international trains. *ContractClient* is used to check whether any special tariffs apply to a customer. For example, big companies can make a deal with the railroad operators, so that reduced fares are used when their employees book tickets. In such cases, the *ContractClient* will find and provide these special tariffs. *OMA* is responsible for processing payments. It delivers the payment screens for several payment providers (e.g., iDeal, Visa, MasterCard and more). Finally, *ITTI,* the most complex system, is responsible for booking a trip. Its behavior is discussed later in this chapter.

One of the major railroad operators has decided to provide a new sales channel, called *SelfService,* enabling customers (mainly business travelers) to book international trips using a generic web browser. For that reason, the operator contracted a software company to build an integration solution, called *Travel Information Provider (TIP).* The purpose of *TIP* is to integrate the existing information system, provide new functionality (currently missing in *DirectMode*), and serve as a basis for the *SelfService* application. In addition, the railroad operator has expressed its wish to replace *DirectMode* client (the Java application) with a web application that also uses *TIP*. Figure 8-143 depicts the desired architecture of the booking system.

[38] This case only concerns *DirectMode* channel

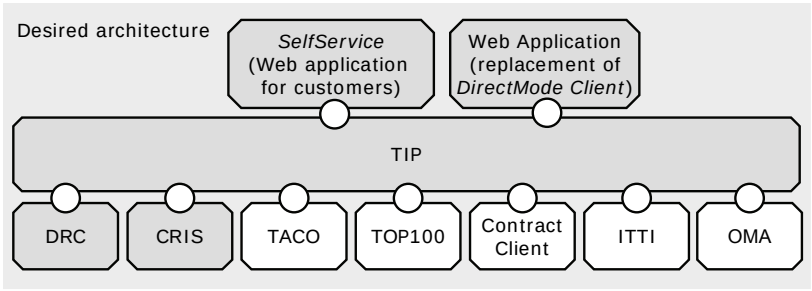


Figure 8-143
Desired situation

The *DirectMode* system lacks a capability to determine the possible routes from A to B. The railroad operator has decided to provide this capability in the *SelfService* application by adding a new system called *DRC*. *DRC* provides a service to determine the possible routes from A to B. Furthermore, since *DirectMode* is only used by the desk employees it does not provide a capability for customers to store their travel preferences and reuse them next time they want to make a booking. The railroad operator has decided to provide this capability in the *SelfService* application by adding a new system, called *CRIS. CRIS* is responsible for managing customer profiles. Amongst others, it provides operations to register a new customer (input customer profile, output acknowledgement) and to check customer credentials.

Currently, *DirectMode* does not show the price unless a booking is made. A customer might ask the service desk employee for the price and then choose not to book at all. If the booking is not cancelled, it will remain in the system, even if the payment and delivery of the booking have not been completed. This can lead to a loss of money, as the booked (unpaid) seat cannot be booked again. To deal with this problem the railroad operator has decided to add a new behavior to *TIP*: *TIP* first should provide the customer with an estimated price based on the applicable tariffs for all segments of the trip. Only if the customer starts the booking process a "provisional booking" will be made. This booking will be cancelled if the final steps (payment and delivery) have not been fulfilled.

In the following sections, we apply the steps of our integration method to specify the services of *TIP*, verify its correctness, and generate an implementation using model transformation.

8.2 Application of the Integration Method

The biggest obstacle in the case study, presented in this chapter, was the limited availability of information about *DirectMode* and the systems it uses. More specifically, the following information sources have been available:

– The *XML schemata* of the messages sent from DirectMode to some of the systems it uses.

– A *test environment* for *DirectMode*. Using the test environment, it was possible (to a certain extent) to analyse the incoming and outgoing messages and to derive their missing XML schemata.

– A *manual* on how *DirectMode* is used. The manual has been used to clarify the semantics of the exchanged messages and to identify the functionality of *DirectMode*.

8.2.1 Steps 1 and 2. Deriving the PIM models

Since the only information we had at our disposal was the information about DirectMode, we had to use this information to derive the (partial) service PIMs of the systems it integrates. Therefore, Steps 1 and 2 (cf. Figure 8-144) of our method were performed manually, analyzing the interactions between DirectMode and its environment.

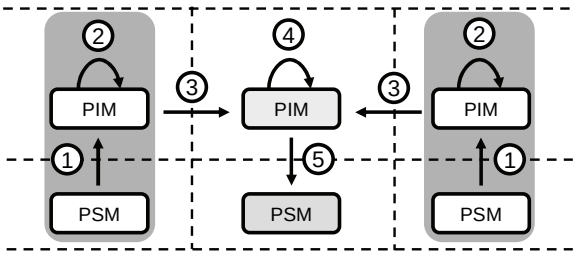


Figure 8-144 Steps 1 and 2. Deriving the (semantically enriched) service PIMs of the systems

In Chapter 4, we showed that COSMO supports modeling services from different perspectives at different levels of abstraction. In this section, we model services of DirectMode starting from integrated perspective at choreography abstraction level. Then we refine the model to a distributed orchestration (see Chapter 4, Section 4.3.2) and this way present the distributed choreography models of the systems used by DirectMode.

We start with the integrated choreography (see Chapter 4, Section 4.3.2) of DirectMode services (cf. Figure 8-145).

The behavior of DirectMode starts with either requesting an existing booking (*retrieveDossier*) or creating a new one (*provideTariffs*). In the first case, there are two options that follow: the existing booking can be either changed (*changeBooking*) or canceled (*cancelBooking*). In the second case, DirectMode provides tariff information (*provideTariffs*) and availability information (*provideAvailability*). If seats for a requested trip are available, DirectMode allows a new booking to be created (*createBooking*). Note that only if this new booking is created, the exact price of the ticket can be

shown (*providePrice*). At this point, the customer can decide to pay for the ticket or save the booking and pay latter. If the customer decides to pay, he is provided with a list of possible payment methods (*getPaymentMethods*). Once he selects one of them, he is presented with the respective payment screen where he can enter his account data and perform the actual payment. Only if the payment succeeds a ticket will be printed and given to the customer.

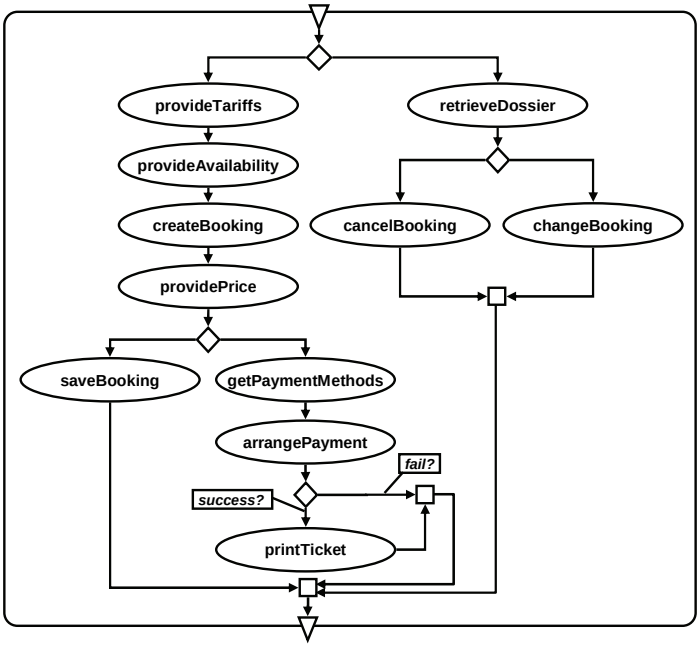


Figure 8-145
The integrated choreography model of DirectMode

In the first refinement step, we assign responsibilities to *DirectMode* and its client. That is, we refine the model presented in Figure 8-145 and derive the distributed choreography model of *DirectMode*. For the sake of simplicity, we only present the refinement of the services responsible for creating a new booking. The resulting distributed choreography model of the *DirectMode* services is presented in Figure 8-146.

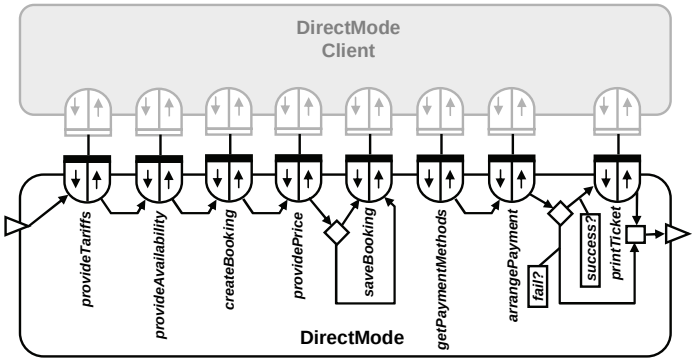


Figure 8-146
The distributed choreography model of DirectMode

In the second refinement step, we model the internal actions performed by DirectMode in order to provide its services. At this point, we are only interested in what actions are required and not in who is responsible to perform these actions. The resulting integrated orchestration model of DirectMode services is presented in Figure 8-147.

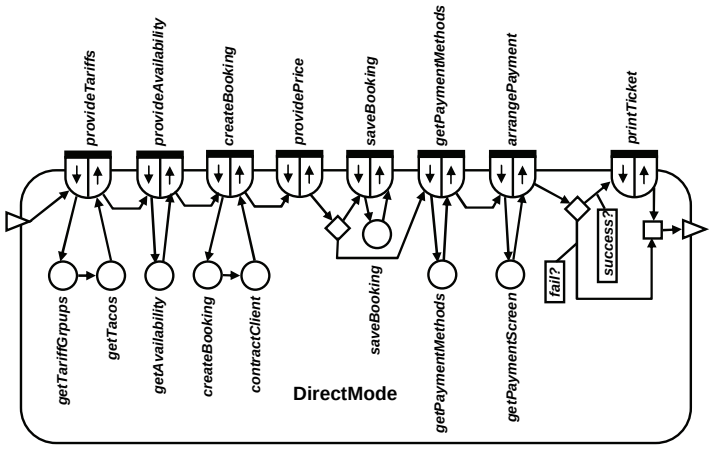
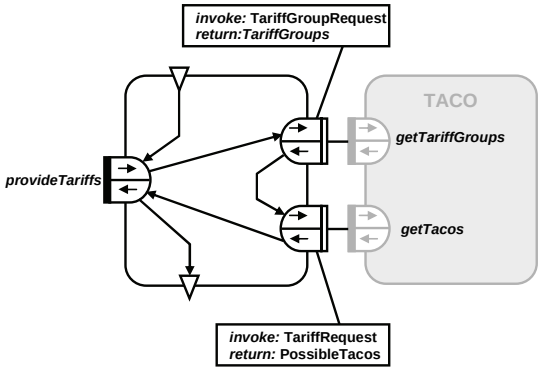


Figure 8-147
The integrated orchestration model of DirectMode

In the final refinement step, we refine each internal action to an interaction by defining the responsibility of each system that participates in this action.

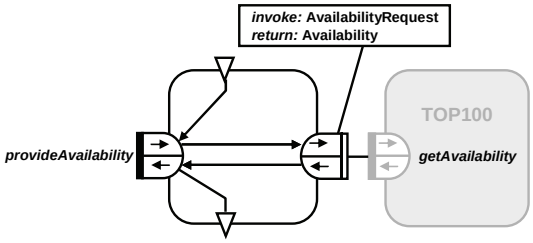
The TACO system is responsible for providing tariff and pricing information. It integrates tariff information from the various inventory systems (used by the various railway operators). In Figure 8-148 we present the refinement of the actions *getTariffGroups* and *getTacos*.

Figure 8-148
Tariffs sub-behavior



TOP100 provides up-to-date availability information for international trains. For a certain train (on a certain date and time) it returns the number of seats available for every tariff that is still available. In Figure 8-149, we present the refinement of the action *getAvailability*.

Figure 8-149
Availability sub-behavior



To create a new booking DirectMode interacts with ITTI and ContractClient systems. ITTI is used to create the actual booking whereas ContractClient system is used to determine the price of the ticket (since special tariffs may apply to some customers). If the customer agrees to pay the price, DirectMode will interact with ITTI to obtain available payment methods (*getPaymentMethods*). After selecting a payment method, DirectMode requests the respective payment screen for that method by interacting with OMA (*getPaymentScreen*) and presents it back to the customer. Upon successful payment, DirectMode client can print the ticket. In Figure 8-150, we present the refinement of the actions *createBooking*, *contractClient*, *saveBooking*, *getPaymentMethods* and *getPaymentScreen*.

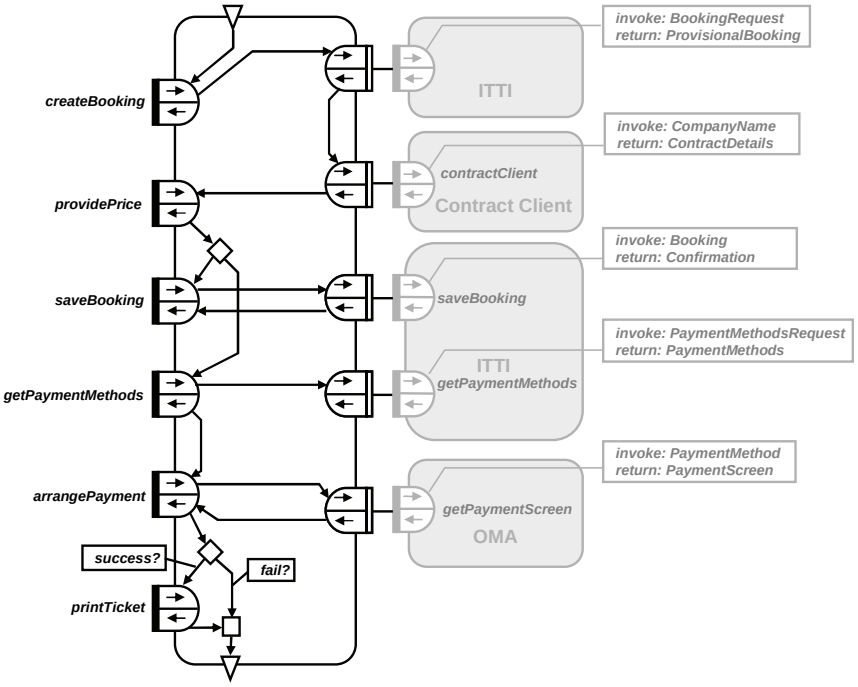


Figure 8-150
Booking sub-behavior

8.2.2 Step 3. Design of TIP framework

In Step 3 of our integration method, we address the new requirements of the railroad operator by specifying the service PIM of *TIP* (cf. Figure 8-151).

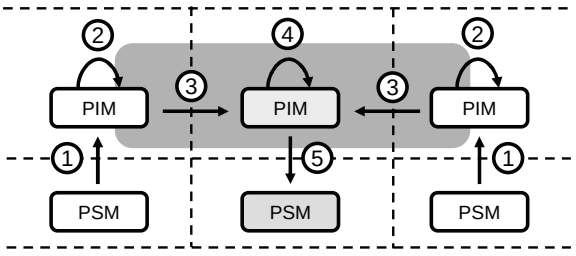


Figure 8-151
Step 3. Design of the TIP

To address the requirements, *TIP* should provide additional services to the *SelfService* application. First, it should integrate a new system called *DRC* and provide a service to determine the possible routes from A to B. Second, it should integrate a new system called *CRIS* and functionality for managing customer profiles. Finally, *TIP* behavior should provide support for

"provisional booking". The integrated behavior of *TIP* is presented in Figure 8-152.

The behavior of *TIP* starts with either a new travel request made from the SelfService or a request to login an existing customer. In case of a new travel request, *TIP* first provides the possible routes from A to B. Then, it provides the possible tariffs and availability for the selected route.

When a route is planned and the customer has logged in successfully a new provisional booking can be created. Following this step, different payment and delivery options are presented to the customer. At this point, he can arrange the payment. If the payment is not arranged within a certain period, the booking will be automatically cancelled. If the payment succeeds, the booking will be either fulfilled or confirmed. Fulfilled booking means that no further action on behalf of the railway companies is required. E.g., customers print their own ticket (home print). Confirmed booking means that the booking is paid for and reserved, but not yet finalised. It will be finalised when the ticket is printed and mailed to the customer.

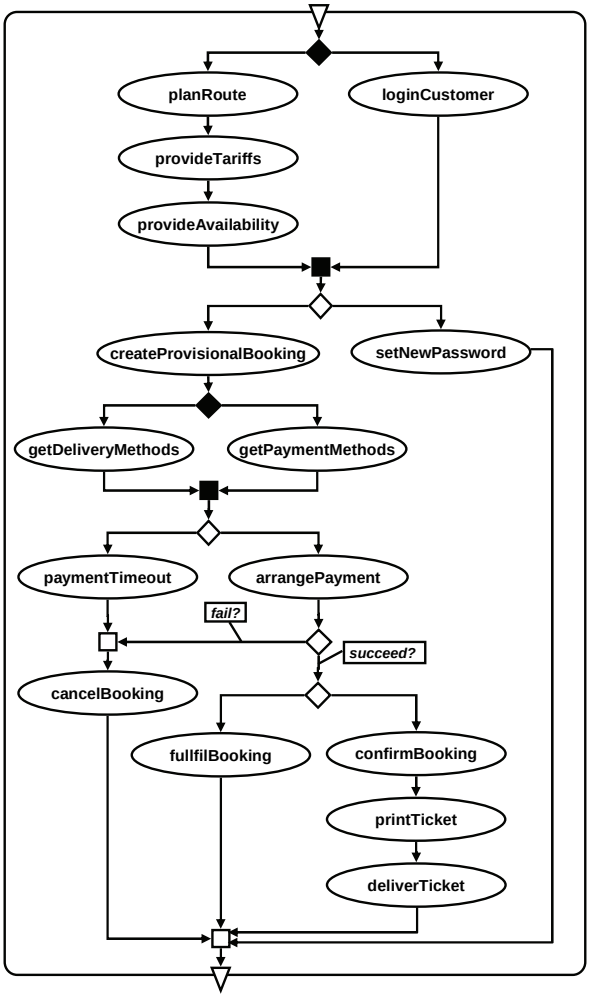


Figure 8-152
The integrated choreography model of TIP

Similar to *DirectMode*, we refine the integrated choreography of *TIP* by assigning responsibilities to *TIP* and *SelfService*. That is, we refine the model presented in Figure 8-152 and derive the distributed choreography model of *TIP*. Again, for the sake of simplicity, we only present the refinement of a part of *TIP*. The resulting distributed choreography model of the *TIP* services is presented in Figure 8-153.

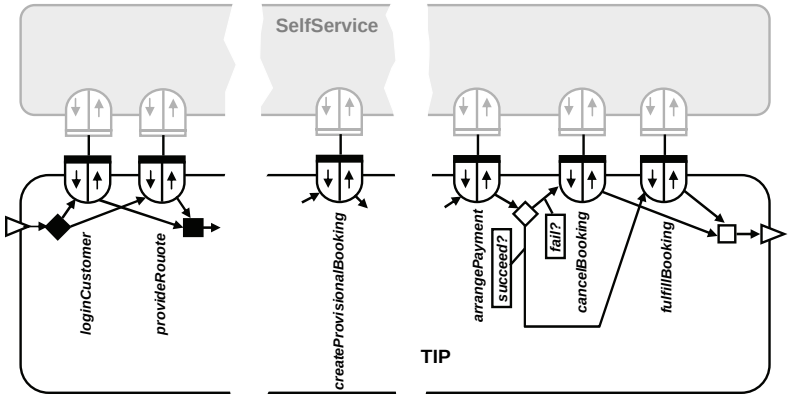


Figure 8-153 The distributed choreography model of TIP

In the second refinement step, we model the internal actions performed by TIP in order to provide its services. This step is analogous to the second refinement step performed on the DirectMode model. For that reason, we skip the presentation of the integrated orchestration model of TIP and continue with presenting the distributed orchestration.

To make a booking, a customer first has to login. If the customer has logged in successfully, the SelfService application will be provided with customer information. Otherwise, an error message will be shown to the customer. When logged in customers may also change their passwords (cf. Figure 8-154).

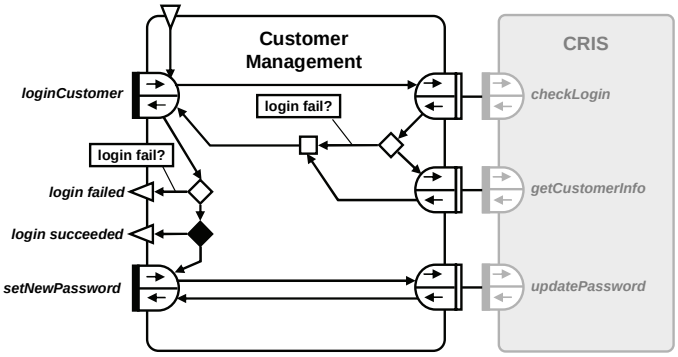


Figure 8-154 Customer Management sub-behavior

To make a booking a customer also has to plan a trip. *TIP* provides this functionality by integrating a new system called *DRC* (cf. Figure 8-155).

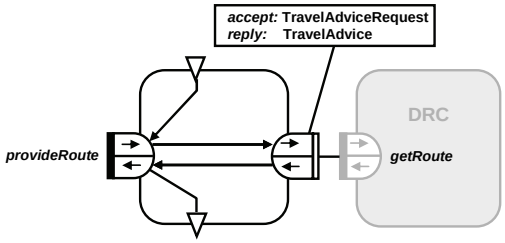


Figure 8-155
Route planning sub-behavior

Only when the customer has logged in and a certain travel option is selected a booking can be made. This is done by executing the behavior presented in Figure 8-156.

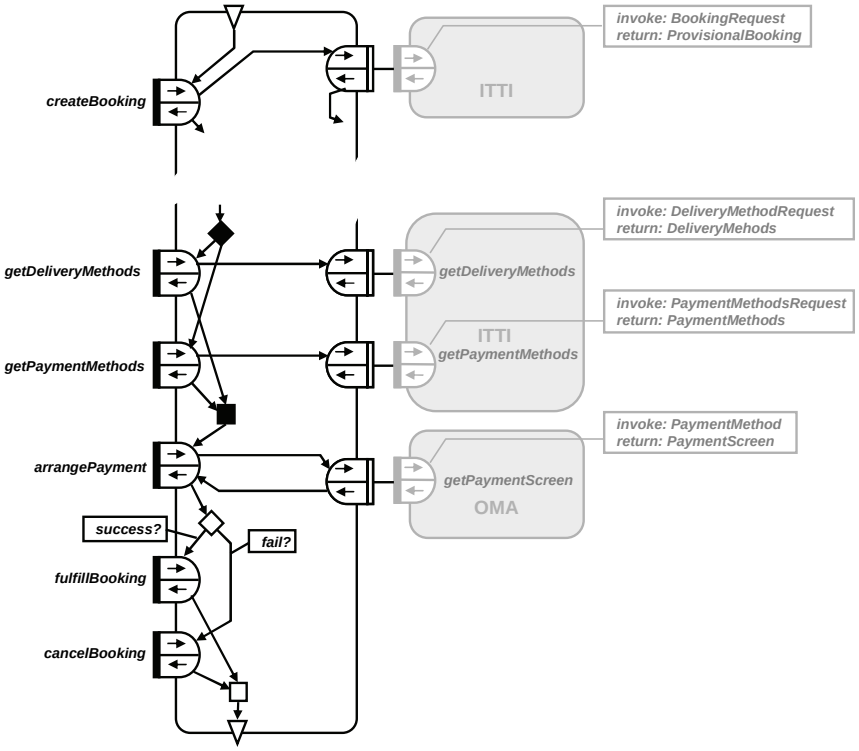


Figure 8-156
Booking sub-behavior

Besides specifying the behavior of TIP, we also needed to specify the mappings between the information models of the systems that TIP integrates and the information model of SelfService application. However, in this case, most of the SelfService operation invocations have been defined to match the operation executions of the systems that TIP integrates. This resulted in a very few mapping definitions. For example, SelfService expects a *TravelOption* that aggregates tariffs and availability information. In this case,

we defined a mapping to combine data from *provideTariffGroups*, *provideTacos* and *getAvailability* operations.

To define the mappings we re-used the mapping DSL presented in Chapter 7.

8.2.3 Step 4. Verification of the integration solution

In Step 4 (cf. Figure 7-129) of our integration method, we verify the service PIM of *TIP*.

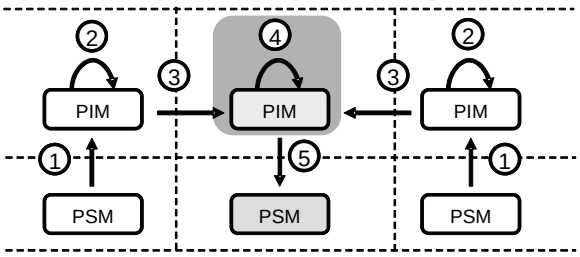


Figure 8-157
Step 4. Verification of the integration solution

To do this, we first abstract from the participation of each system and construct the integrated behavior of TIP. In fact, we already have the integrated model of TIP (cf. Figure 8-152). Next, we transform the integrated behavior of TIP to a Petri Net and construct the respective occurrence graph. Then, we use the occurrence graph to check whether the integrated systems are pragmatically interoperable (see Chapter 5, Section 5.3.3). Finally, we analyse the occurrence graph to check whether TIP meets the new requirements.

This step is identical to Step 4 presented in Chapter 7. In fact, we reuse the COSMO to Petri Net transformation, which we developed for the SWS Challenge case study. Figure 8-158 presents the resulting Petri net.

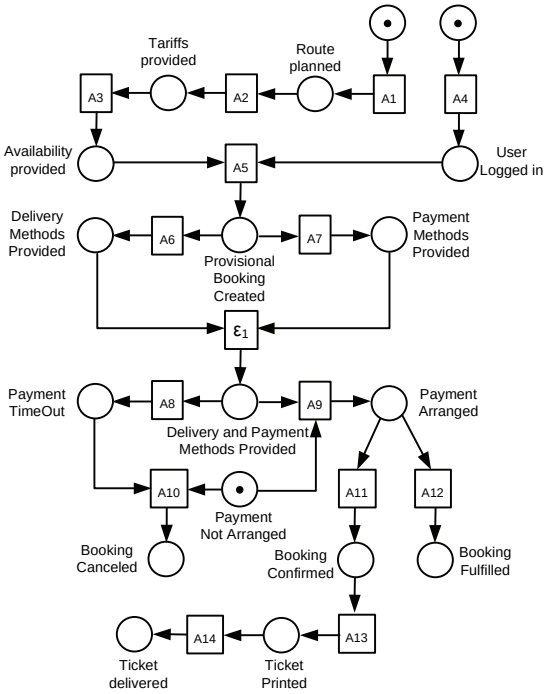


Figure 8-158
The Petri net corresponding to the integrated choreography of TIP

The table below presents the correspondence between the actions from the integrated model and the transitions in the resulting Petri nets.

A1	planRoute
A2	provideTariffs
A3	provideAvailability
A4	loginCustomer
A5	createProvisionalBooking
A6	getDeliveryMethods
A7	getPaymentmethods
A8	paymentTimeout
A9	arrangePayment
A10	cancelBooking
A11	confirmBooking
A12	fulfillBooking
A13	printTicket
A14	deliverTicket

Next, we construct the occurrence graph of the net (cf. Figure 8-159)

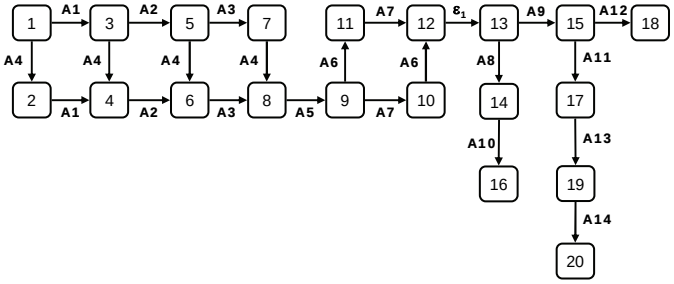


Figure 8-159
The occurrence graph of the net from Figure 8-158

Similar to Step 4 in the previous chapter, we analyse the occurrence graph to check whether the integrated system supports all desired execution traces. For example, we check whether a booking can be fulfilled (cf. Figure 7-133)

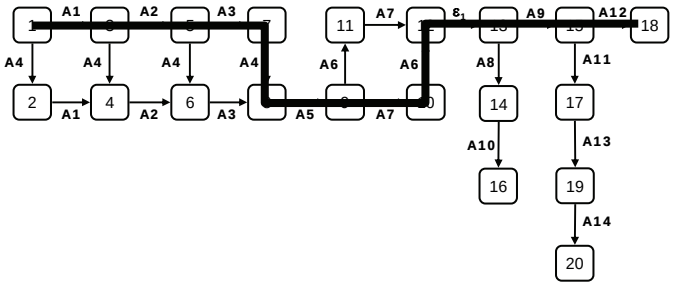


Figure 8-160
Checking whether a booking can be fulfilled

The answer is "yes" because there is an execution trace from A1 (*planRoute*) and A4 (*loginCustomer*) that leads to A18 (*fulfillBooking*).

Another interesting query is to check whether a ticket can be printed before it is paid (cf. Figure 8-161).

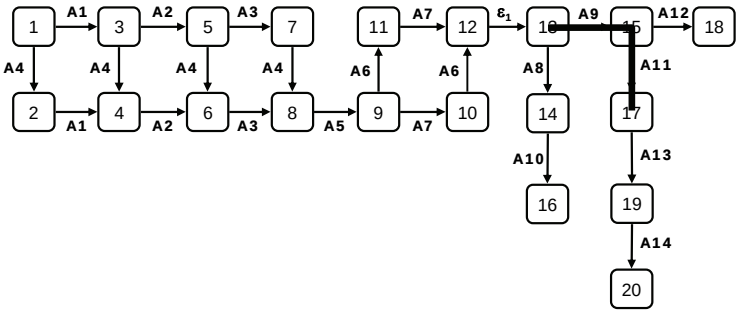


Figure 8-161
Checking whether a ticket can be printed before paid for

The answer is "no", because the only execution path that leads to A11 (*printTicket*) is via A9 (*arrangePayment*).

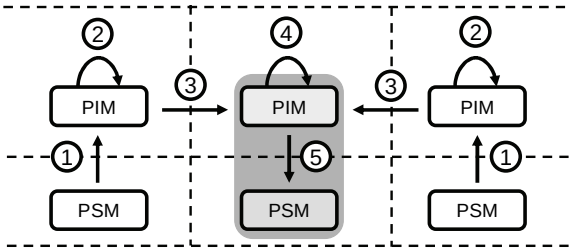
Figure 8-159, Figure 8-160 and Figure 8-161 only serve for illustration purpose. In real life, we do not need to visualise the net and occurrence

graph. In fact, we only need to compute the occurrence graph, represent it in some format (e.g., as a relational database), and query it.

8.2.4 Step 5. Deriving the service PSM of TIP

In Step 5 (cf. Figure 8-162) of our integration method, we transform the service PIM of *TIP* to a PSM in terms of Java.

Figure 8-162
Step 5. Deriving the service PSM of TIP



Similar to the case presented in Chapter 7, the control flow patterns from service PIM of the *TIP* system are recognised and then transformed to a pattern-oriented service model in terms of *Common Behavioral Patterns Language* (CBPL). As aforementioned, each CBPL pattern represents a control flow that is common to most execution languages, i.e., *sequence*, *concurrence*, *selection* and *iteration*. Besides, message sending and receiving, data transformations and constraint evaluation activities in the CBPL model are replaced with operations to interact with the *Data Flow Manager* and the integrated systems. Finally, the CBPL model is transformed to a PSM in terms of Java. For that purpose, we developed a simple process execution engine and defined a transformation that creates a configuration for the engine.

The metamodel of our process execution engine is show in Figure 8-163.

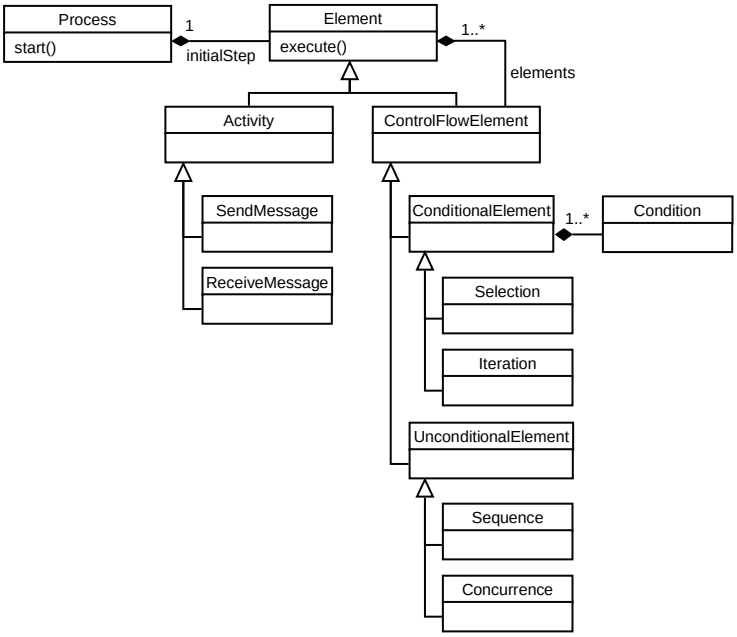


Figure 8-163
The meta-model of the process execution engine

A *Process* consists of executable *elements* that are either *activities* or *control flow elements*. An *activity* can be either *sending* or *receiving* a message. After a message is received, the process engine interacts with the *Data Flow Manager* to update its state. Before sending a message, the process engine interacts with the *Data Flow Manager* to retrieve the information required to construct the message. At this stage, all required data transformations are performed by the Data Flow Manager.

The class *Sequence* implements the CPBL pattern *sequence*.

```
public class Sequence
      extends UnconditionalControlFlowElement {

   ...

      @Override
      public void execute() {
         for (Element element : this.getElements()) {
            element.execute();
         }
      }
}
```

In the *execute()* method of this class we iterate over all sub-elements and execute them sequentially.

The class *Concurrency* implements the CBPL pattern *concurrency*.

```
public class Concurrency extends
      UnconditionalControlFlowElement {

      ...

      @Override
      public void execute() {
         CountDownLatch doneSignal =
            new CountDownLatch(this.getElements().size());
         for (Element element : this.getElements()) {
            new Thread(new Worker(element, doneSignal)).start();
         }
         try {
            doneSignal.await();
         } catch (InterruptedException e) {}
      }
}
```

In the *execute()* method of the class we iterate over all sub-elements and start a new thread for each of them. To synchronise the execution of the threads, we use a *CountDownLatch*.

A *CountDownLatch* is initialised with a given count. In our case, this is the number of the sub-elements of the respective *Concurrency* object. The *await()* method of the *CountDownLatch* blocks until its count reaches zero. The count is decreased by invoking the *countDown()* method. Once the count reaches zero all waiting threads are released and the execution flow continues immediately.

To provide a mechanism for decreasing the count of the *CountDownLatch* we wrap the sub-elements of the *Concurrency* object in a special class *Worker*. The *Worker* is a simple class that implements the interface *Runnable*. In Java, the *Runnable* interface should be implemented by any class whose instances are intended to be executed by a thread. The implementing class must define a method *run()* with no arguments. In this method, the wrapped process element is executed and, upon successful completion, the count of the associated *CountDownLatch* is decreased by calling the method *countDown()*.

```
public class Worker implements Runnable {

      private final CountDownLatch doneSignal;

      private Element element;

      public Worker(Element element, CountDownLatch doneSignal)
{
         this.element = element;
         this.doneSignal = doneSignal;
      }

      public void run() {
         element.execute();
         doneSignal.countDown();
      }
```

```
}
```

The class *Iteration* implements the CBPL pattern *iteration*.

```
public class Iteration
      extends ConditionalControlFlowElement {

      ...

      @Override
      public void execute() {
            while (this.getCondition().check(this.getContext()))
               this.getElement().execute();
      }
}
```

In the *execute()* method of the class we repeatedly execute a process element until the controlling condition evaluates to *false*. The evaluation of the condition is delegated to the Data Flow Manager and is based on the current state of the system.

The class *Selection* implements the CBPL pattern *selection*.

```
public class Selection extends
      ConditionalControlFlowElement {

      ...

      @Override
      public void execute() {
         for (int i = 0; i < this.getElements().size(); i++) {
            Element element = this.getElements().get(i);
            Condition condition = this.getConditions().get(i);
            if (condition.check(this.getContext())) {
               element.execute();
               break;
            }
         }
      }
}
```

In the *execute()* method of the class we check whether the condition, associated with each sub-elements holds, and, if so, we execute this element.

Each CBPL model is transformed into a configuration specification for the process engine. For that purpose, we use the Spring Framework[39]. The key component of Spring is its *Inversion of Control container*[40], which provides a consistent means of configuring and managing Java objects.

At run time, Spring reads the configuration specification, instantiates all required objects and injects all specified dependences among them.

[39] http://www.springsource.org/

[40] http://static.springframework.org/spring/docs/2.5.x/reference/beans.html

```
public class TIP {
   public static void main(String[] args) {
      BeanFactory factory = new XmlBeanFactory (
         new ClassPathResource("TIPConfig.xml"));
      Process process = (Process) factory.getBean("process");
      process.start();
   }
}
```

8.3 Deriving the Service PSM of TIP in Terms of WS-BPEL

To demonstrate that our integration method allows for changes of the implementation technology we present a hypothetical scenario, in which railroad operator decides to use WS-BPEL as implementation technology for *TIP*. A reason for using WS-BPEL might be to take advantage of more scalable and feature-rich execution engines which enable logging and monitoring of process execution.

To generate the WS-BPEL process from the service PIM of *TIP*, we re-use the transformation presented in Chapter 7. In this section, we present an approach for exposing the systems used by *TIP* as Web Services.

As aforementioned in Chapter 3, a WS-BPEL process interacts with other systems via WSDL interfaces. However, the systems used by *TIP* do not provide such WSDL interfaces. Therefore, we first need to expose the functionality of these systems as Web Services.

In our approach, we use *Apache Axis WSDL2Java*[41] tool. The tool takes as input a Java interface and produces a respective WSDL description. However, we do not have the Java interfaces of the services used by TIP. Fortunately, the distributed orchestration model of TIP provides all the information required to generate these interfaces automatically. The transformation is trivial – for each system, we generate a Java interface that contains methods corresponding to all operation executions defined in the service PIM of that system. The arguments of these method have the same types as the results established in the *accept* part of the operation execution. The return types of the methods are the same as the respective result types established in the *reply* part of the operation execution. The transformation is illustrated in Figure 8-164.

[41] http://ws.apache.org/axis/

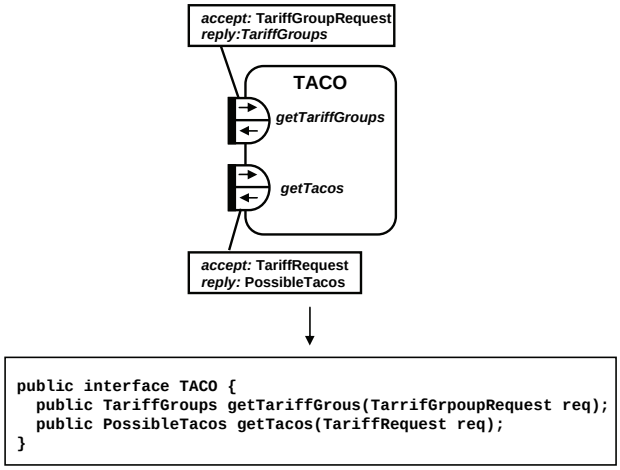


Figure 8-164 Deriving the Java interface of TACO system

Once we have the java interfaces of all systems, we use Java2WSDL [42]tool to produce the respective WSDL descriptions. The tool requires as input information that is not present in the service PIMs of the systems, such as the location of the service and the target namespace of the WSDL file. This information is provided manually.

The Java2WSDL tool outputs a WSDL document that contains the appropriate WSDL *types*, *messages*, *portType*, *bindings* and *service* descriptions to support a SOAP RPC encoding. If the specified methods in the Java interfaces reference other classes, the Java2WSDL tool will generate the appropriate XML types to represent the classes and any nested/inherited types.

In the next step, we use the generated WSDL to produce all of the glue code for deploying the service as well as stubs for accessing it. For that purpose, we use the Apache Axis's WSDL2Java tool. The tool takes as an input the base output directory for java classes, the scope of deployment (Application, Request, or Session) and the name of the WSDL file and produces a number of Java classes. For example, for the WSDL generated in the previous step the outputted files are:

– *TACOSoapBindingImpl.java - the implementation code for the Web service*

– *TACO.java – a remote interface to the TRC system*

– *TACOService.java – the service interface of the Web services. The TACOServiceLocator class (see next bullet) implements this interface.*

[42] http://ws.apache.org/axis/

- *TACOServiceLocator.java – a helper factory for retrieving a handle to the service*
- *TACOSoapBindingSkeleton.java – a server-side skeleton code*
- *TACOSoapBindingStub.java – a client-side stub code that encapsulates client access*
- *deploy.wsdd – a deployment descriptor that we pass to the Axis system to deploy these Web services*
- *undeploy.wsdd –a deployment descriptor that we use to undeploy the Web services from the Axis system.*

We focus on the generated *TACOSoapBindingImpl* class:

```
public class TACOSoapBindingImpl {
  private TACOImpl taco;
  public TACOSoapBindingImpl(TACOImpl taco) {
    this.taco = taco;
  }
  public TariffGroups getTariffGrous(TarrifGrpoupRequest req) {
    return taco.getTariffGrous(TarrifGrpoupRequest req);
  }
  public PossibleTacos getTacos(TariffRequest req) {
    return taco.getTacos(TariffRequest req);
  }
}
```

When the Web Service is deployed and the *TACOSoapBindingImpl* is instantiated it is initialised (via its constructor) with the *DirectMode* implementation, responsible for the actual communication with the *TACO* system. At run-time, when Axis receives a SOAP message designated for *TACO* system, it will unmarshal it and invoke the respective method of the *TACOSoapBindingImpl* class. The *TACOSoapBindingImpl*, in turn, will delegate the invocation to the *DirectMode* implementation.

To initialise the *TACOSoapBindingImpl* with the implementation of the DirectMode class responsible for the interaction with *TACO* system, we define the following Spring description:

```
...
<beans>
   <bean id="DirectModeTACO" class="DirectModeTACO">
   </bean>

   <bean id="TACOWebService" class="TACOSoapBindingImpl">
     <constructor-arg>
        <ref bean="DirectModeTACO"/>
     </constructor-arg>
   </bean>
```

```
...
</beans>
```

Finally, the web services and the WS-BPEL process are deployed and can be executed.

8.4 Summary

In this chapter, we applied our method for the semantic integration of service-oriented applications presented in Chapter 5 in a concrete context. I.e., we solved a concrete integration problem from the real world using concrete SOA, MDA and KR technologies. Note, that we solve the case using real-world data but in lab settings, i.e., our solution has not been deployed and used in the real-world.

Similar to the case presented in Chapter 7, when applying the method we observed a number of effects. The observations supported the validity of the claims made in Chapter 6 and this way, validated the that our method meets the requirements defined in Chapter 1.

Chapter 9

Discussion

In this chapter, we discuss the validation results for our integration method presented in Chapter 5. This chapter is organised as follows: Section 9.1 provides a discussion about the validity of the claims made in Chapter 6. In Section 9.2, we provide a cross-case analysis of the two cases presented in Chapters 7 and 8. Section 9.3 summarises the most important challenges that emerged when applying our integration method to the cases as well as lessons learnt. Finally, in Section 9.4 we discuss identified limitations of our solution.

9.1 Validation Claims

In Chapter 6, we made a number of claims to validate whether or not our integration method meets the requirements presented in Chapter 1. In this section, we provide arguments for the validity of these claims. These arguments are based on observations we made when solving the integration problems presented in Chapters 7 and 8.

To solve the integration problem of Scenario 1 (cf. Chapter 7), we developed a model transformation that takes a service PSM (specified in WSDL) and produces a corresponding service PIM (in terms of COSMO). We applied this transformation to derive the service PIMs of the systems to be integrated. By contrast, when solving the integration problem presented in Chapter 8, we did not have explicit service PSMs in machine processable format. To derive the service PIMs (see Chapter 8, Section 8.2.1) we had to experiment with the existing integration solution, read its manual and go through its source code. Nevertheless, in both cases we were able to derive the service PIMs of the systems and to confirm Claim C1 (*Service PIMs can be derived from service PSMs*). For the case presented in Chapter 7, the derivation process was fully automatic, whereas for the case we presented in Chapter 8, the service PIMs were derived manually.

In both cases, we used COSMO concepts to model the service PIMs of the solutions of the respective integration problems. In Chapter 7, we used the distributed choreography PIMs of the systems to be integrated. First, we mirrored the PIMs and automatically derived the initial service PIM of the integration solution. Then, we manually refined this initial service PIM and derived the complete service PIM of the integration solution. In Chapter 8, we started from a higher-level, i.e., we first specified the integrated choreography PIM of the integrated solution. Then, in a number of refinement steps, we derived the complete PIM of the integrated solution. In both cases, COSMO provided all necessary concepts to model the service PIMs of the integrated solutions. This, in turn, has confirmed Claim 2 (*COSMO provides all required concepts to model platform-independent integration solutions*). In addition, regarding the case from Chapter 8, COSMO abstraction layers and perspectives helped us to address only a limited set of concerns in series of design steps. In this way, we were able to focus on issues that have been relevant for each of these steps while ignoring or discarding details that have been irrelevant for the same step.

In Chapter 7, we used OWL to map the elements of the service PIMs of the systems to be integrated to classes and properties from UDEF (in this case UDEF trees have been used as domain ontology). This way, we provided shared, agreed upon semantics for these model elements. Besides, a WSDL description only defines the services *provided* by some system and does not define the services *requested* by it. Using COSMO, we formally specified both services requested and provided by the systems. Finally, WSDL does not provide means for specifying the ordering of service operations. Using COSMO, we formally defined the order of service operations. Thus, by applying the semantic enrichment step, in which we added additional information and behavior semantics to the service models of the systems to be integrated, we were able to make these service descriptions more explicit and less ambiguous. By doing this, we have confirmed Claim 3 (*The service models of the systems to be integrated can be semantically enriched*).

In Chapter 7, we used OWL to define the information model of the integration solution. This allowed us to aplly a formal logical reasoner to check the *Necessary Conditions 1* and *2*. To validate necessary condition 3, we developed a model transformation from COSMO to Petri Nets. In Chapter 7 and 8, we transformed the behavior models of the integration solutions to Petri Nets and constructed the respective occurrence graphs. Based on the occurrence graphs were able to check whether or not the integrated systems satisfy *Necessary Condition 3*. Being able to check the necessary conditions for interoperability we confirmed Claim C4 (*The necessary conditions for interoperability can be formally checked*).

In Chapter 7, Section 7.3, we presented a second scenario for which the integrated requirements have changed. We showed that only updating the service PIM of the integration solution was sufficient to generate an implementation reusing the model transformation developed for the first scenario. By doing this, we confirmed Claim C6 (*The same model transformations can be used to solve different integration problems*).

Finally, to confirm Claim C5 *(The same solution PIM can be used to derive different solution PSMs)*, in Chapter 8, Section 8.3 we presented an hypothetical variation of the integration problem from Section 8.2. In this variation, the requirements on the implementation technology have changed. To address the new requirements, we developed a new model transformation and applied it to the same PIM of the integration solution presented in Section 8.2. This way, we showed that the same abstract solution could be reused to derive an implementation for the new technology platform.

9.2 Cross-case Analysis

In this section, we analyse the cases and identify commonalities and differences between them. By doing this, we seek to provide further insight into the practical applicability of our conceptual framework and integration method in more general context.

The integration problems, presented in Chapters 7 and 8 originate from *different domains*. In the first case, we built an integration solution for a problem from the order management domain. In the second case, we applied our integration method to solve a characteristic problem from the travel domain. Based on experiences with both cases, we expect that our conceptual framework and integration method can be used in *different domains* to model a wide spectrum of services. When defining the conceptual framework and the integration method, in addition, no assumptions have been made with respect to the type of services that should be modeled and integrated.

To derive the service PIMs of the systems from Chapters 7 and 8, we used completely *different sources of information*. In the first case, there were explicit service descriptions in WSDL. Although defining only the syntax of the exchanged messages, these descriptions served as starting point to derive the initial service PIMs of the systems. In the second case, we had no explicit service descriptions. To derive the service PIMs we had to experiment with the existing integration solution as well as to analyse its documentation and source code. As already said, this is a very long and difficult process. Nevertheless, in both cases we were able to derive the service PIMs of the systems. In the first case, we automated the process by

providing a model transformation from XML schema to OWL. In the second case, we had to do this manually. Having an explicit service description, in a machine processable format, enables the quick initialization of a service PIM. Such a raw PIM can be presented in a suitable way (e.g. using a graphical tool) to the business domain experts who can specify the semantics of the model elements.

Another difference between the two cases is the nature of the *integration problem*. In the first case, we had a "*green field*" situation, i.e., there was no existing integration solution. We had to analyse the mismatches between the systems and design an integration solution that enables them to interoperate. In the second case, there was already an existing integration solution. Our task was to change this solution such that it continues to enable interoperability between the systems while meeting the new integration requirements. In the first case, we specified the integration solution by first identifying the mismatches in the information and behavior models of the systems, and then providing a solution for each identified interoperability problem. In the second case, we were able to reuse some fragments of the existing integration solution and only had to provide solutions such that the new integration solution meets the new integration requirements.

In the second step of our method, we allow service PIMs, derived in the first step of the method, to be semantically enriched. One way to do this is to map their elements to elements of domain ontologies with well-defined and shared meaning. In the first case, we were able to do this by mapping the classes and properties of the information PIMs to objects and properties defined in UDEF. In the second case, we were unable to find suitable domain ontologies. Nevertheless, in this second case, the information model of the application that uses the integrated services has been designed to match as closely as possible the information models of the integrated systems. This resulted in definition of much simpler mappings comparing to the first case.

9.3 Challenges and Lessons Learnt

One of the biggest challenges, solving the integration problems in both cases, was to understand the language used in the problem domains. Although the provided XML schemata could be easily transformed into information models, understanding and explicitly defining the meaning of the model elements was a very hard task. For example, *Blue* exchanges messages in RosettaNet's PIP3A4 format. Although the XML schema of PIP3A4 seemed relatively simple at first sight, a closer look revealed that it imports another XML schema, defining large amount of RosettaNet core

elements. In addition, the imported XML schema, in turn, imports another large XML schema, defining various vocabulary terms such as country codes, order status codes and currency codes. Since we are no domain experts, it was difficult to understand the meaning of all elements, defined in these XML schemata. Besides, most of the elements defined in the XML schemata, have been nested in deep structures and referenced from other elements. This also contributed to the complexity of the information models. A tool capable of visualizing the information model in a graphical way and providing a mechanism for showing/hiding parts of the model would help significantly to get a quick overview of the model elements and their structure. However, we could only guess the meaning of these elements. To understand their semantics we needed a domain expert.

Another challenge was the lack of information about the systems in the second case. Discovering what a system does by experimenting with the system, reading its manual and going through the source code, is an extremely difficult process. It could be avoided if the services of the systems would be properly described at development time.

Specifying and verifying the mappings between the information models of the systems from the first case, also turned out to be a very complex task. The reason for that was that we did not have a tool supporting our mapping DSL. To specify the mappings, we used a generic text editor, which resulted in many syntax errors. Finding and repairing these syntax errors took most of the time required to solve the integration problem. Even a very simple tool, e.g. a text editor with auto-completion function, could improve dramatically the process of specifying the information mappings.

Finally, a big challenge was dealing with conflicting sources of information about the systems in the second case. For example, the user manual of DirectMode has been written before the document, describing the design of the system. For that reason, there were mismatches between these two information sources. In addition, the XML schemata, used to specify the format of the exchanged messages did not always match the actual implementation of the message exchanges.

Applying our integration method to the two cases led to a number of lessons learnt.

First, describing the services of a system is very important. Reasoning with service descriptions is already very difficult even without having to go through the source code or user manuals of a system.

Second, in order to reuse fragments an existing integration solution, the service PIMs of these solutions should be made as modular as possible. This means specifying the PIMs as a composition of (logically independent) components. Our conceptual framework is well-suited for this providing two different ways for structuring, namely constraint-based and causality-based structuring.

Second, specifying only the syntax of the messages, exchanged between the systems is not enough. Reasoning about a system requires richer semantic descriptions about the exchanged messages and the functionality of the system. In some cases, there were very useful comments in the source code of the systems and the XML schemata files. However, it would be much more useful if these comments were captured in a formal way in a service description.

Another lesson learned is that routine comes from practice. After starting with simple COSMO models, it became less difficult and time-consuming to define more complex models. Besides, in both cases, modeling systems helped us to understand what they do.

The last but not the least important lesson is that the modeling work is not to be underestimated. Trying to understand what a complex system does, especially when not much information is available, takes large amount of time. This makes the planning of such a task very hard.

9.4 Limitations

In Step 1 of our method, we transform the service PSMs of the systems to be integrated to their respective PIMs. However, in some cases, it might be not possible to preserve the complete semantics of a PSM. For example, an XML schema could define the order of children elements within a parent element. Knowledge representation languages in general do not provide means for specifying such an ordering. A possible workaround for this problem would be to explicitly define the missing semantics of PSM elements in dedicated PIM elements.

In the case study from Chapter 7, we used OWL as language to represent the information modeling concepts of our framework. Although providing means for semantic integration by increasing the precision of the service models and having mapping constructs, OWL has limited expressivity that is inherited in our approach. First, OWL does not allow defining predicates of arbitrary arity, formulating complex queries (beyond subsumption and instance checking), and "negation-as-failure" (i.e., closed world assumption) reasoning. Second, it does neither have string manipulation functions (e.g., concatenation) nor arithmetic primitives (e.g. multiplication) and aggregation and grouping (e.g., the sum of all values of a given property) are not supported. Confusingly, these have been defined as objectives in the OWL "Use Cases and Requirements" document (OWL, 2004), but are not present in the current version of the language definition.

Although using KR languages to express the mappings between the information models of the systems to be integrated, our method requires system integrators to discover and represent these mappings. This is a

manual, error-prone process which requires understanding of the meaning of all information models being integrated. In some cases, wrong mappings can cause interoperability problems that cannot be discovered by satisfiability reasoning.

In addition, it is not realistic to assume that system integrators are familiar with KR languages, such as OWL. This means that tools are required to hide the complexity of OWL and guide the system integrators in defining the mappings. To take full advantage of the computational properties of OWL, such tools could integrate an OWL reasoner and provide feedback about the correctness of the mapping.

In our integration method, we check the third necessary condition for interoperability by constructing the integrated service model, transforming it to a Coloured Petri net, and analyzing the occurrence graph of the net. The basic idea is to compute all reachable states and state changes of the integrated system and represent these as a directed graph. The advantage is that the approach is fully automatic and allows for the verification of many properties of the integrated system by querying the graph. The main disadvantage is that in some cases the state space may explode.

To automate the translation of COSMO models to Coloured Petri nets we specified a model transformation in QVT. However, the mappings between the COSMO concepts and the concepts of Coloured petri net have not been formally verified.

In the final step of our method, we transform the service PIM of the integration solution to service PSM in terms of some implementation technology (e.g., WS-BPEL or Java). In most cases, the PIM is semantically richer than the target PSM. Therefore, not all PIMs can be transformed to PSMs. A way to deal with this problem would be to define a language profile to restrict the semantics of the PIM elements such that they match the one of the target PSM elements. In some cases (when a PIM element cannot be mapped to a single PSM element) such a profile should also provide composition rules. Similar to the first step of our method, the specification of the model transformation is a complex, manual (and therefore, error-prone) process, which requires knowledge about the metamodels of both service PIM and PSM. Nevertheless, change in the implementation technology will only require change in the transformation specification; the same service PIM of the integration is used as source of the new transformation. Hence, the integration design is preserved.

Part V.
Conclusions

Chapter 10

Conclusions

This chapter presents the conclusions of this thesis and identifies some topics for future research. The chapter is organised as follows: Section 10.1 presents a summary of our work. Section 10.2 presents our main research contributions. Section 10.3 reflects on the properties of our integration method. Finally, Section 10.4 provides recommendations for future research.

10.1 Summary

Enterprise application integration is an extremely complex process because it has to deliver a solution that compensates for *differences* amongs multiple heterogeneous, autonomous and distributed systems in order to enable their *interoperability*. Therefore, to build a correct integration solution, a system integrator must understand what *interoperability* means and what possible mismatches there could be between information systems.

In Chapter 2, we studied existing definitions of interoperability and identified their common characteristics. Based on this analysis, we provided our own definition of interoperability and identified three different levels of interoperability, namely *syntactic*, *semantic* and *pragmatic* interoperability. Syntactic interoperability is outside the scope of this thesis. Therefore, we focused on *semantic* and *pragmatic* interoperability. Analyzing various literature sources from different areas such as artificial intelligence, database research and process integration, we identified possible interoperability problems at semantic and pragmatic level.

There are many existing integration tools available nowadays. The contribution of our research to the users of these tools is that we provide a *conceptual model of the EAI problem*. This model will help them to better *understand* the *underlying concepts* and *problems.* Understanding the concepts and awareness of the possible problems will enable system integrators to

make more informed and carefully thought-out design decisions. Furthermore, the identified concepts will provide them with a *vendor independent vocabulary.* Such a vocabulary will enable efficient communication between system integrators and problem stakeholders. Finally, we hope that our research will inspire tool vendors and standardization bodies to shift their attention from syntactic to semantic and pragmatic interoperability problems.

In Chapter 3, we studied existing enterprise application integration approaches and identified their drawbacks. In addition, we presented the most significant emerging technologies that promise to improve these approaches. We concluded that existing integration approaches and technologies are too technical for business domain experts. This hinders their participation in the integration process and requires software engineers to take decisions beyond their competence. In addition, existing integration solutions are not flexible, i.e., they cannot easily deal with changes in business requirements or with changes in implementation technology.

Service orientation, model-driven development and various knowledge representation technologies have emerged to improve existing enterprise application integration approaches. However, these emerging technologies address different problems of the existing integration approaches. None of them eliminates the identified drawbacks completely. For that reason, we proposed to combine particular elements of these technologies in Chapter 5. In Chapter 3, we argued which elements are useful and what are the possible interactions among them.

In Chapter 4, we studied existing definitions of the term "service". In analyzing them, we identified common characteristics of services. We used these common characteristics to define a conceptual framework for service modeling called COSMO. Using this framework, one can model the domain of a system, its services and their relations. Based on this one can reason whether these services are interoperable. For example, in our integration method, we used COSMO to represent the service models of systems that needed to be integrated. In addition, we used COSMO to design the service models of the integration solution.

In Chapter 5, we proposed a model-driven method for the semantic integration of service oriented applications. Our method provides solutions for each of the interoperability problems identified in Chapter 2. In a number of steps the method defines how to build end-to-end integration solutions.

In the first step, we derive platform independent service models of the systems to be integrated by abstracting all technical details from their platform-specific service models. In the next step, we increase the coverage and precision of derived platform independent models by adding additional

semantic information that cannot be derived from their platform-specific models. In the third step we solve the integration problem at a technology-independent level. This enables the more active participation by the domain experts. In addition, the semantically-enriched service models allow some integration tasks to be fully or partially automated. In addition, the abstract nature of the integration solution allows its reuse for different implementation technologies. In the next step, we formally verify the correctness of the integration solution using simulation and automatic reasoning. In the final step of the method the platform independent service model of the integration solution is transformed to a platform-specific service model by adding technical details specific to the implementation technology.

In Chapters 6 to 9, we validated our integration method by applying it in a particular context, using concrete technologies. In Chapter 1, we identified a number of *requirements* for integration methods in general. To verify whether our method meets these requirements we made a number of claims and provided arguments for their validity. We did this by applying our method using concrete technologies to solve two integration problems from the order management domain and the travel domain, respectively. When applying our integration method we observed a number of effects. We analysed our observations and argued to what extent our integration method meets the requirements defined in Chapter 1.

10.2 Research Contributions

The research, presented in this thesis, contributes to the area of enterprise application integration. Our main contributions are the following:

- We identified *common characteristics of interoperability* and gave a definition of *interoperability*. Next, we identified three different levels of interoperability, namely, *syntactic*, *semantic* and *pragmatic* interoperability. At each of these levels, we identified *possible interoperability problems*. Awareness of the possible interoperability problems enables system integrators to make more informed and carefully thought-out design decisions. In addition, the identified problems served as input to design our service integration method, i.e., we analysed the problems and provided solution for each them.

- We analysed commonly found interpretations of the service concept and identified *general service properties*. We introduced and illustrated basic concepts that support the modeling of these properties and underly the service concept. Using these basic concepts we explained, related and

formalised important notions, such as service *choreography* and *orchestration*. Our conceptual framework consists of a *small number of basic concepts*, based on practice and provides a powerful conceptual basis for service modeling. It is *language-independent*, but at the same time its basic concepts can be related to many of the popular languages used in the context of service design, analysis and implementation. This opens the possibility to use the framework as *common semantic model* for comparing and analyzing models specified in different languages. Our conceptual framework is *domain-independent*, i.e., no assumptions are made with respect to the type of systems for which services should be modeled. The framework is suitable for wide spectrum of application domains, e.g., it can be used to model services at a business, application and component level, thus beyond the usual domain of web services. Finally, the framework supports the modeling of services at *different abstraction levels*. More precisely, we identified three generic abstraction levels, namely, service *effect*, *choreography* and *orchestration*.

- We identified *necessary conditions* for interoperability and proposed a *method for verifying* whether a set of systems are interoperable. Our verification method enables the early discovery of false agreements and the automatic verification of integration solutions. This, in turn, results in reduced cost and time to deliver the end solution.

- We proposed a *method for the semantic integration of service-oriented applications*. The key feature of our method is that *semantically-rich* service models at *different abstraction levels* are employed to develop flexible *integration solutions*. Our method provides solutions for each of the identified interoperability problems. In addition, the method defines all steps for building integration solutions from business requirements to software realization. Last but not least, the method allows for changes in the implementation technology as well as for changes of business requirements.

10.3 Reflection

In Chapter 1, we defined requirements for integration methods. In this section, we discuss to what extent our method meets these requirements.

- *Requirement R1.* The method should provide for defining the integration solutions in terms of the problem domain, rather than in terms of solution technologies. To meet this requirement we have defined a conceptual framework for service modeling that is technology

independent (cf. Chapter 4). Our concepts represent general service properties identified by analyzing service definitions from different domains such as economics, business science, telecommunications and software engineering. In addition, our conceptual framework provides for modeling services at different levels of abstraction. In this way, a solution can be defined closer to the problem domain and refined to an implementation in terms of a concrete technology by adding technology-specific details. The refinement is captured in formally defined transformations which provides for traceability of the design choices and enables the conformance checks. In some domains, COSMO concepts might be too generic or non-intuitive. Using transformations one can define a DSL and relate its concepts to concepts (or combinations of concepts) from COSMO. In this way, business experts will be able to describe their requirements and review the proposed solution using terminology they are familiar with. This will also shield them from the formal semantics of our concepts while they can still take advantage of the analytical features provided by our framework.

– *Requirement R2*. The integration method should enable the semantic integration of services. Existing service description standards merely provide languages to specify the syntax of the messages exchanged between systems. Although, this enables machines to check whether data in the messages conforms to a specific syntactic schema, it entirely leaves to humans the task of interpreting and using these data. In many cases, data is ambiguous and can be misinterpreted, which, in turn, leads to undesired effects of their use. Our integration method enables the semantic integration of different services by allowing formal knowledge representation techniques to be used to specify the information models of the systems. In this way, data in the exchanged messages can be automatically checked not only for syntactic but also for semantic correctness. In addition, our method allows system integrators to relate their information models to shared, agreed-upon, domain-specific ontologies reducing further the ambiguity of the exchanged data.

– *Requirement R3*. The integration method should enable the formal verification of the integration solution. Currently, the correctness of an integration solution is verified by implementing it and performing tests on the implementation. In this way, incorrect solutions are discovered at a very late stage resulting in increased cost and time to deliver the end solution. To address this requirement, we allow information models of the systems to be specified using a formal knowledge representation language. Further, we defined necessary conditions for interoperability

and a method to check whether a number of systems are interoperable given an integration goal. Note, that in some cases formalizing an information model can be a very difficult task or a formal verification might not be required. For that reason, this step in our method is optional.

– *Requirement R4.* The integration method should allow for *changes in the implementation technology.* This means that if the implementation technology changes, it should be possible to reuse the same abstract solution specification defined by the domain experts. To address this requirement, in our method we provide a step in which a new model transformation can be developed and applied to the same abstract solution. In this way, a new implementation is generated automatically reusing the knowledge captured in the abstract model of the solution. A slight variation of this step would be to develop a transformation to a different language with analytical capabilities and perform additional verifications of the same abstract solution.

– *Requirement R5.* The integration method should allow for changes of the business requirements. This means that if the business requirements change, only the abstract solution specification has to be updated to reflect the new business requirements. It should be possible to generate a solution implementation from the updated abstract solution specification. This is an important requirement, because enterprises constantly change their systems and services to address new market demands. We addressed this requirement by capturing the abstract solution in a technology independent model and providing a generic, domain independent transformation from COSMO to the respective implementation technology.

10.4 Future Work

To enable the adoption of our results by the industry there are still some issues that need to be further investigated. In this section, we summarise them and provide some direction for a future research.

Tooling. The adoption of our integration method is largely determined by the existence of tools that not only guide system integrations in the design of the integration solution but also automate their work. For example, such a tool could provide a library of generic building blocks that can be customised and assembled together to build a solution to a concrete integration problem. Further, the tool could provide a library of reusable

model transformations from different service description languages to COSMO and vice versa. This will not only increase the value of the framework, but will also help to further validate its concepts by the means of concrete cases.

Automatic Construction of the Integrated Solution. At this moment, we construct the integration solution manually. First, we manually specify the mappings between the information models of the systems to be integrated. Then, we mirror all interaction contributions of the systems together with their conditions and manually add the causality relations among the interaction contributions. Even with a good tool, these manual activities can be expensive and error-prone. For that reason, it is desirable to automate the process of constructing the integrated solution. To achieve this, two main research tasks have to be performed: First, an investigation is needed how to use existing approaches for automatic ontology mapping to derive the information model of the integration solution. Second, an investigation is needed how existing automatic service composition approaches can be used to derive automatically the relations among the interaction contributions of the integration solution.

Analytical Features. COSMO concepts have a formal foundation as discussed in Chapter 4. This enables automatic reasoning about certain properties of the services described by these concepts. We used these feature of our conceptual framework to provide a method for the automatic verification whether a number of systems are interoperable. However, using formally defined concepts presents more possibilities for automatic reasoning. For example, by assigning an execution time to an activity one can reason about the performance of the integration solution.

Non-functional Properties. In our current work, we only briefly mentioned the non-functional aspect of services. However, often non-functional properties play an important role in the process of designing, implementing and managing the integration solution. For example, to use the services of some system, the integration solution may need to implement certain security or transaction protocols. Likewise, in some cases the integration solutions may need to perform an activity within a certain time frame.

Besides technical properties such as security and response time, our framework could be extended to provide concepts to model the business properties of the services. We consider the value provided by some service to be the most important business property. Further research is needed to align our conceptual framework with existing frameworks for business modeling such as e3value (Gordijn and Akkermans, 2001).

Appendix A

Mapping COSMO to Petri Nets

Constructing and analyzing the state space of a system is a complex process that requires sophisticated algorithms and tools. For that reason, we present a mapping from COSMO to Coloured Petri Net (CPN) (Jensen, 1992) (Jensen, 1994). This way, we can reuse existing tools and take advantage of the knowledge and the best practices developed by the Petri Nets community.

A classical Petri net consists of a set of *places* (represented by circles), a set of *transitions* (represented by blocks), *directed arcs* connecting places to transitions or transitions to places, and *markings* assigning one or more *tokens* (represented by black dots) to some places. CPNs extend the classical Petri nets by providing a mechanism for associating a value of a certain type to each token. In addition, a transition can be enabled only if its input tokens satisfy certain conditions (called *guards*) and can produce output tokens that represent new values (called *bindings*). In this way, a transition can be seen as a function that maps input values to output values in a certain context.

An *action* in our language maps to a *transition* with two *places* as shown in Figure A-165.

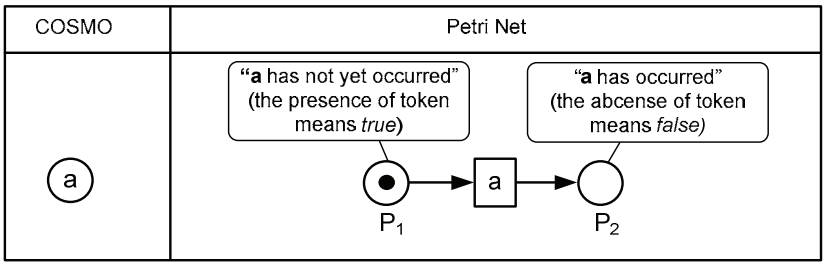
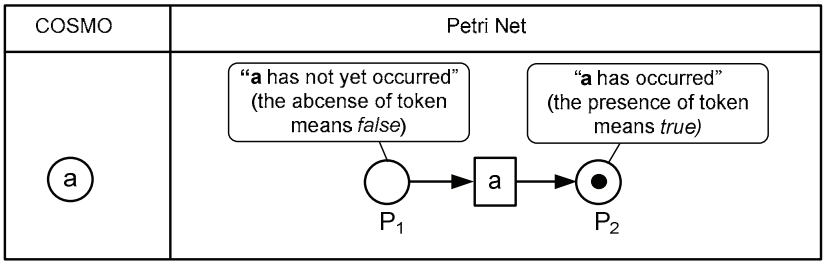


Figure A-165 Mapping an action to transition

The place P_1 represents the condition $\neg a$ (i.e., the action *a* has not yet occurred) and the place P_2 represents the condition *a* (i.e., the action *a* has occurred). The presence of a token in a place means that the respective

condition holds, i.e., a token in P_1 means that the condition "*a has not yet occurred*" is *true* and the absence of a token in the place P_2 means that the condition *"a has occurred"* is *false*. If the action *a* occurs, the respective transition *a* fires, consumes the token from P_1 and produces a token in P_2 (cf. Figure A-166)

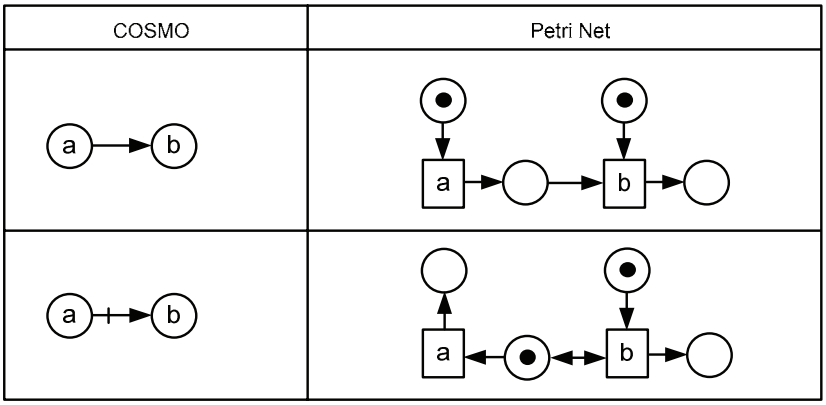
Figure A-166
A marking representing that the action a has occurred



Now, P_1 does not contain a token which means that the condition *"a has not yet occurred"* is *false* and P_2 contains a token which means that the condition *"a has occurred"* is *true*.

The *enabling* relation between two actions *a* and *b* is mapped to an arc from the place, representing that *a* has occurred to the transition *b*. The *disabling* relation between two actions *a* and *b* is mapped to two arcs – one from the place, representing that *a* has not yet occurred to the transition *b* and one from the transition *b* back to the place representing that *a* has not yet occurred. The second arc is necessary because the occurrence of *b* should not change the conditions of *a*. The mapping of the enabling and disabling relations is shown in Figure A-167.

Figure A-167
Mapping COSMO enabling and disabling relations to Petri Net



More complex conditions (for example, conjunction and disjunction of conditions) are mapped in a similar way as shown in Figure A-168. For brevity, we omit the places that are not relevant for the respective example.

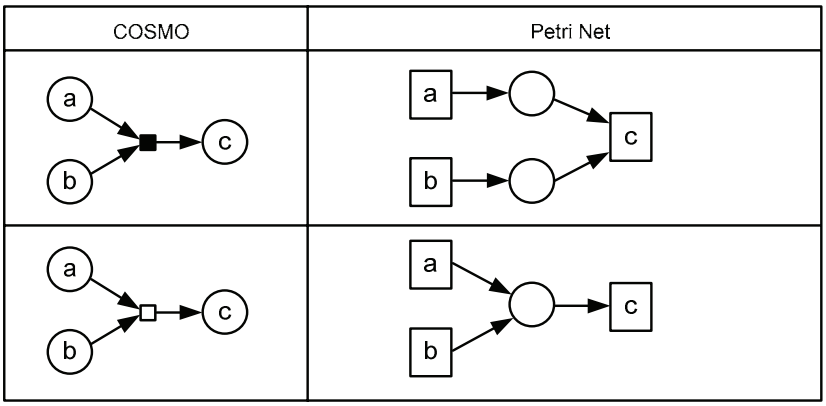


Figure A-168 Mapping conjunction and disjunction of conditions

For completeness, we present the mapping of the operators *and-split* and *or-split* in Figure A-169.

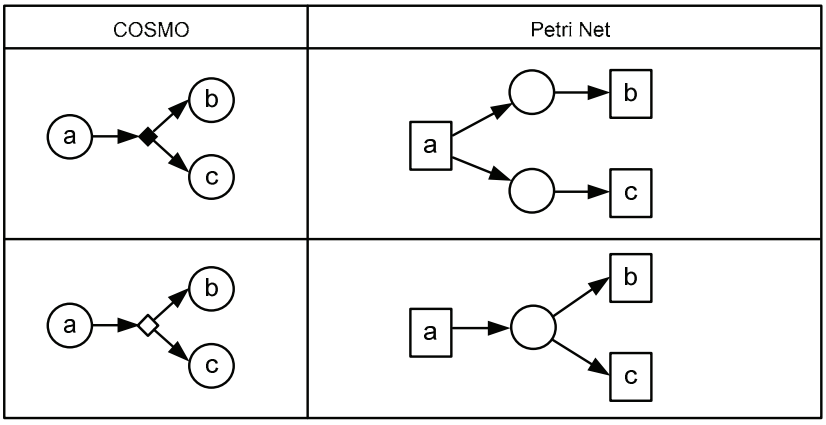


Figure A-169 Mapping and-split conjunction and disjunction of conditions

Each COSMO behavior can be expressed in disjunctive normal form (DNF), i.e., as a disjunction of conjunctions of enabling and disabling conditions. The mapping of a disjunction of conjunctions requires an introduction of a transition that does not correspond to an action in the original COSMO model. An example is shown in Figure A-170. The transition ε can only fire if and only if both transitions *a* and *b* have already fired.

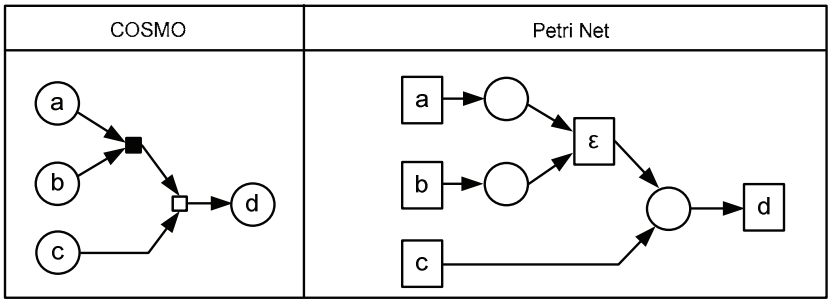


Figure A-170 Mapping a behavior in DNF to Petri Nets

Introducing a ε transition causes a problem when there are disabling conditions in some conjunction that participates in a disjunction. To illustrate the problem, consider the mapping shown in Figure A-171.

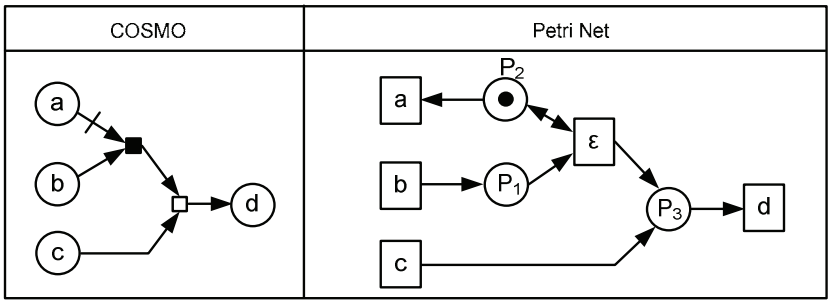


Figure A-171 Wrong mapping of disabling condition in DNF

The COSMO model in the Figure A-171 defines that the action *d* can occur if *c* has already occurred or *b* has occurred and *a* has not yet occurred. We apply the mapping rules defined earlier in this chapter and construct a Petri net as shown in Figure A-171. However, this net does not define the same behavior as the one defined in the corresponding COSMO model. Suppose the transition *b* fires and produces a token in P_1. Next, ε fires (because it has tokens in all incoming places), produces a token in P_3 (representing that *a* has not yet occurred and *b* has occurred) and produces a token back in P_2. Now, suppose that the transition *a* fires (there is a token in P_2); the transition *d* can still fire because there is a token in P_3. However, this is wrong behavior, because *d* cannot occur after *a* has occurred and *c* has not occurred. To deal with this problem we define a *transaction monitor* that blocks the transition *a* until *d* fires.

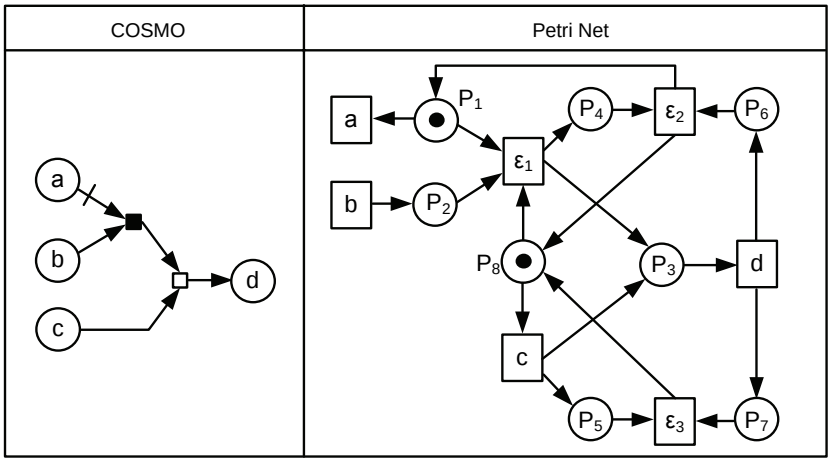


Figure A-172 The correct mapping of disabling condition in DNF

Suppose, *b* fires producing a token in P_2. Next, ε_1 fires, consumes the token from P_1 (and therefore blocks *a*), consumes the token from P_8 (and therefore blocks *c*), produces a token in P_4 (marking the beginning of the transaction) and finally, produces a token in P_3 (enabling *d* to fire because *b* has fired and *a* has not yet fired). When *d* fires it produces a token in P_6 enabling ε_2 to fire. Next, ε_2 fires and commits the transaction by producing tokens in P_1 (enabling *a* to fire) and P_8 (enabling *c* to fire). This way, *a* and *c* cannot fire until *d* commits the transaction. Now, suppose that *c* fires first, consumes the token in P_8 (and therefore blocks ε_1), produces a token in P_5 (marking the beginning of the transaction) and finally, produces a token in P_3 (enabling *d* to fire because *c* has fired). When *d* fires it produces a token in P_7 enabling ε_3 to fire. Next, ε_3 fires and commits the transaction by producing tokens in P_8 (enabling ε_2 to fire).

As said earlier in Chapter 4, the occurrence or the result of an activity may depend on the result of one or more causal predecessors. Such dependences can be easily mapped onto *guards* and *bindings* in terms of CPNs (cf. Figure A-173).

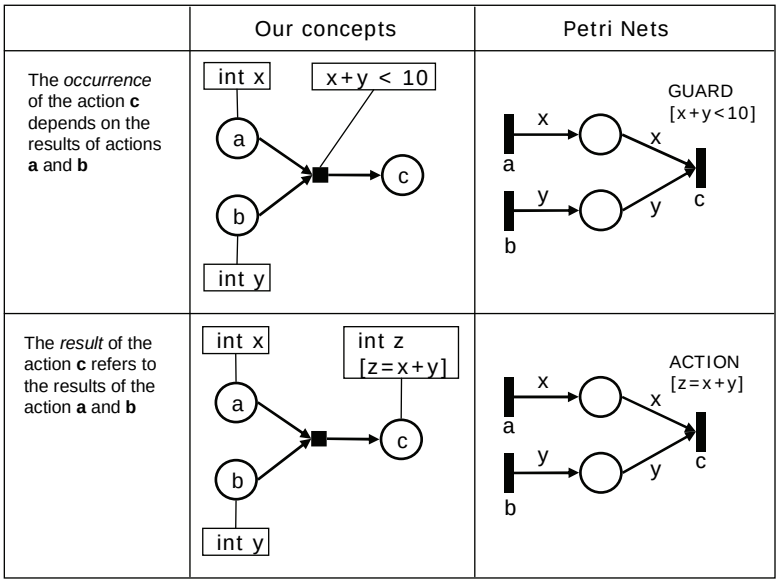


Figure A-173
Mapping to Coloured Petri Nets

The presented mappings allow any model expressed in COSMO to be translated into a CPN and analysed using existing tools. For example, to check whether an integrated system meets necessary condition 3, we translate its model to a corresponding CPN and then construct and analyse the state space graph of that net.

A state space graph for a CPN is a directed graph, with node for each reachable marking and an arc for each occurring *binding element*. A binding element is a pair of a transition and a binding. The source of each arc is the marking in which the associated binding element occurs and the destination of the arc is the marking resulting from the occurrence of that binding. An state space graph of a CPN allows checking properties such as *reachability*, *deadlock-freedom* and *liveness*. In our concrete situation to verify the necessary condition 3, we first check for the existence of markings in which the results defined by all participating systems are established. Next, we check whether these results can be established in an order that meets the causality constraints of all participating systems. The second is done by performing a reachability analysis on the respective state space graph.

To illustrate the verification of necessary condition 3 we use the example described in Chapter 4 (cf. Figure A-174).

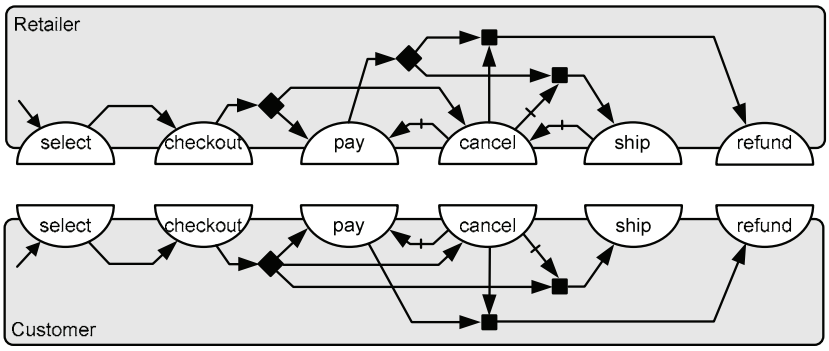


Figure A-174 Example of an online computer shop

To recapitulate, in this the example, a customer has specified the following relations among the interaction contributions of the desired service:

- *select* does not depend on any other activities and can occur from the beginning of the behavior
- *checkout* can only occur if *select* has already occurred
- *cancel* can occur only if *checkout* has already occurred
- *pay* can only occur if *checkout* has already occurred and *cancel* has not yet occurred
- *ship* can occur only if *checkout* has already occurred and *cancel* has not yet occurred
- *refund* can occur only if both *pay* and *cancel* have already occurred

Likewise, the retailer has specified the following relations among the interaction contribution of the provided service:

- *select* does not depend on any other activities and can occur from the beginning of the behavior
- *checkout* can only occur if *select* has already occurred
- *cancel* can occur only if *checkout* has already occurred and *ship* has not yet occurred
- *pay* can only occur if *checkout* has already occurred and *cancel* has not yet occurred
- *ship* can occur only if *pay* has already occurred and *cancel* has not yet occurred
- *refund* can occur only if both *pay* and *cancel* have already occurred

First, we construct the integrated choreography of the service by abstracting from the responsibilities of each participating system. This is done by transforming all interactions into actions and adding all causality relations among them (cf. Figure A-175).

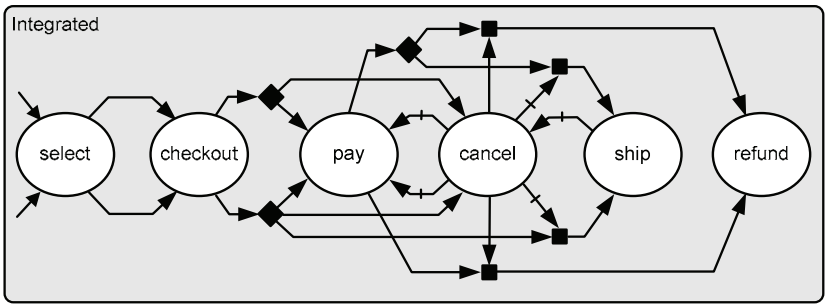


Figure A-175
The example from Figure 5-69 after abstracting from the participating systems

In the next step, we remove all redundant relations (for example, repeating enabling or disabling relations)

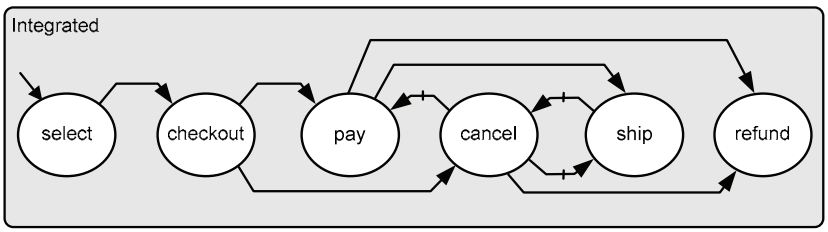


Figure A-176
The example from Figure A-176 after removing the redundant relations

Once we have the integrated service choreography we transform it to a CPN (cf. Figure A-177) using the rules presented earlier in this section.

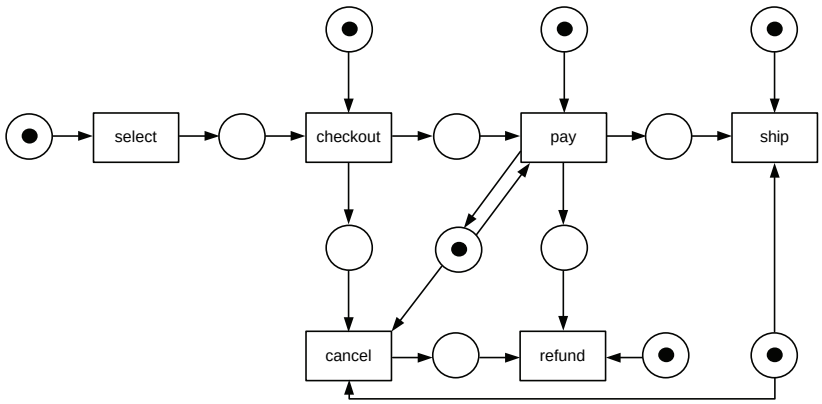


Figure A-177
Possible markings of the example in Figure 5-69

We use the resulting net to construct the respective state space graph (cf. Figure A-178)

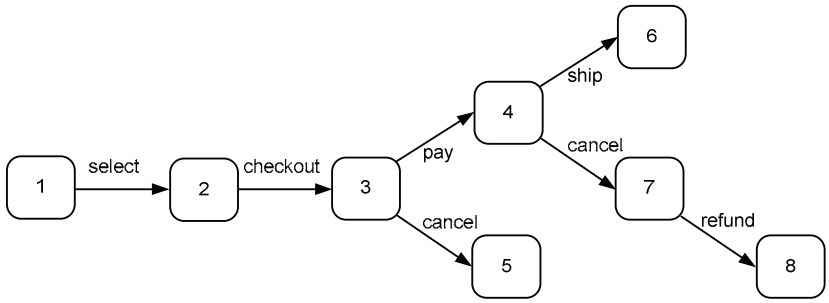


Figure A-178
State space graph of the net in Figure A-177

Now, we can perform a reachability analysis on the state space graph to check whether there is a scenario in which all the service interactions can occur.

Now, suppose that customer has slightly different constraints on the interaction with the retailer as shown in

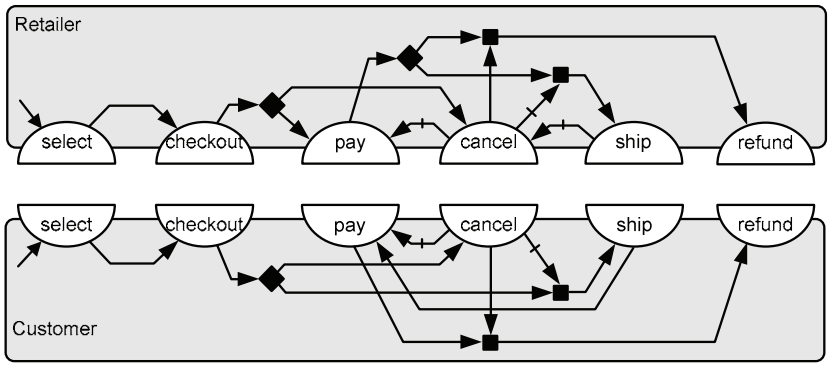


Figure A-179
Modified example

- *select* does not depend on any other activities and can occur from the beginning of the behavior
- *checkout* can only occur if *select* has already occurred
- *cancel* can occur only if *checkout* has already occurred
- ***pay* can only occur if *ship* has already occurred and *cancel* has not yet occurred**
- *ship* can occur only if *checkout* has already occurred and *cancel* has not yet occurred
- *refund* can occur only if both *pay* and *cancel* have already occurred

The integrated service choregraphy of the example from Figure A-178 is shown in Figure A-180.

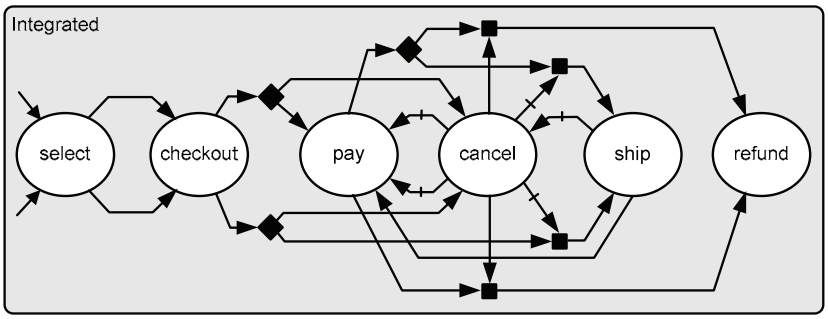


Figure A-180
The example from Figure A-179 after abstracting from the participating systems

After removing the redundant relations, we come to the integrated service choreography shown in Figure A-181.

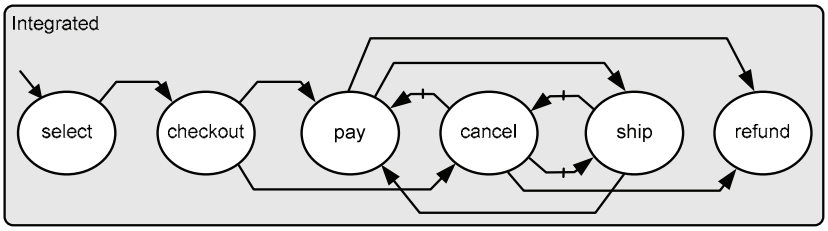


Figure A-181
The example from Figure A-179 after removing the redundant relations

Once again, we transform it to a CPN (cf. Figure A-177)

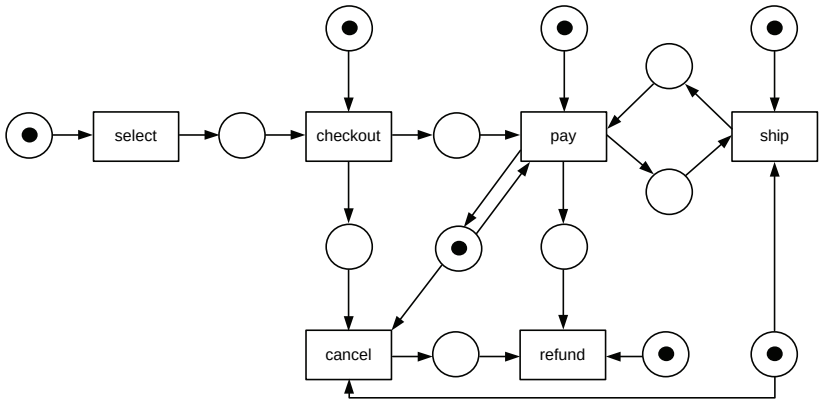


Figure A-182
The net corresponding to the behavior in Figure A-181

We use the resulting net to construct the respective state space graph (cf. Figure A-183).

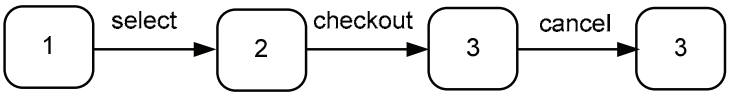


Figure A-183
The state space graph of the net shown in Figure A-182

As one can see, the only possible execution scenario is *select-checkout-cancel* which means that the results of the interactions *pay*, *ship* and *refund* can never be established.

Appendix B

The XML Schemata of SWS Challenge Case

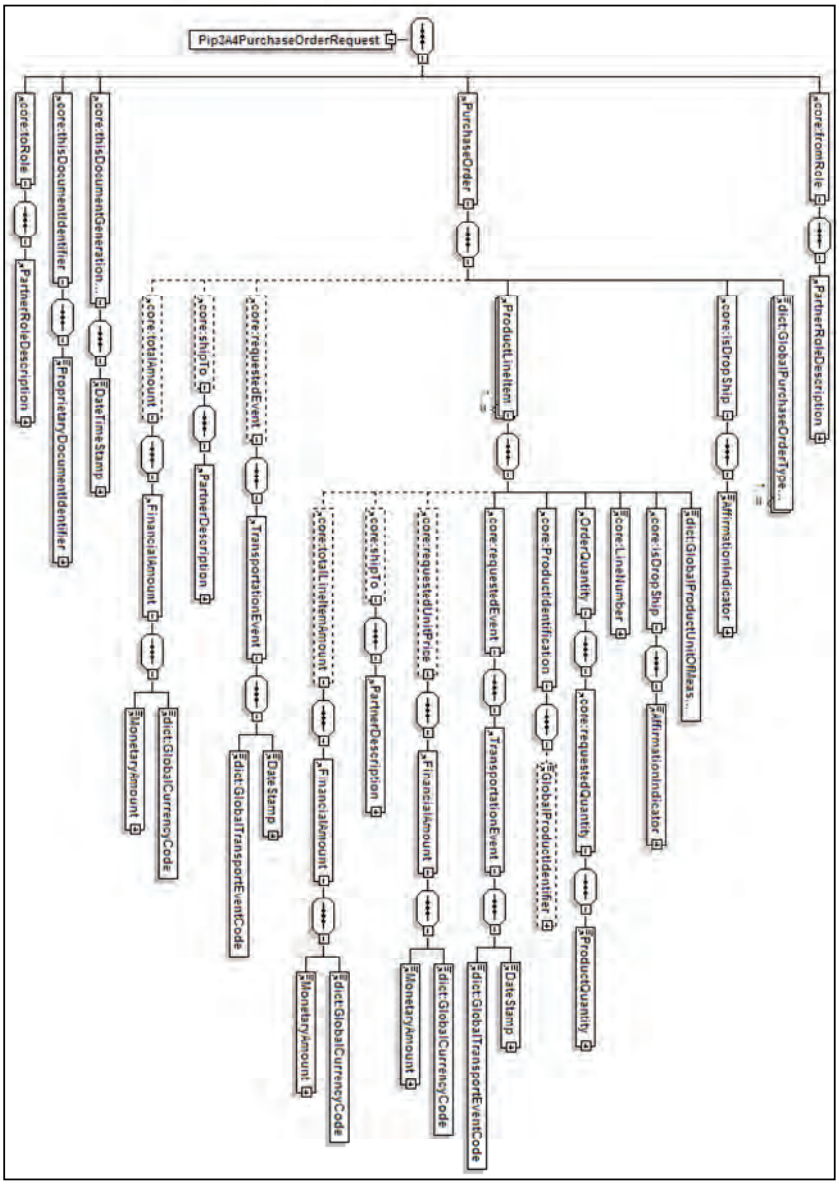


Figure B-184
XSD of message M_1 PIP 3A4 Purchase Order Request messages

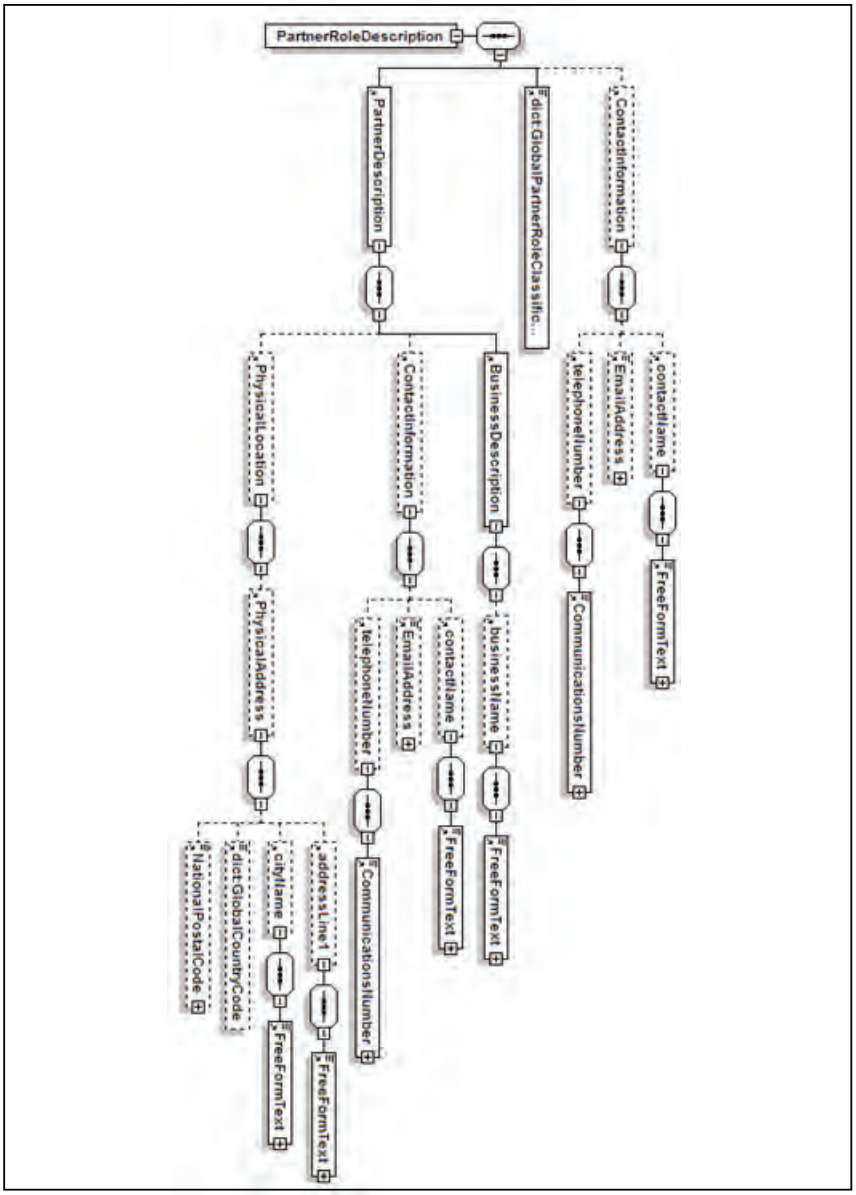


Figure B-185
XSD of message M_1 PartnerRole Description part

Figure B-186
XSD of message M_2 Acknowledgement of Receipt

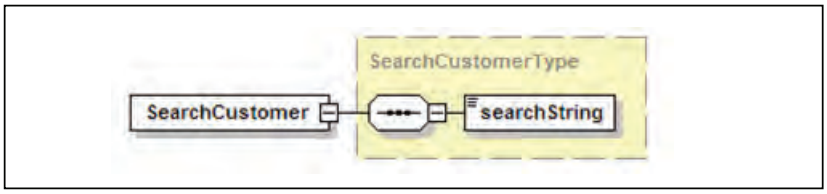


Figure B-187
XSD of message M_3 Search customer Request

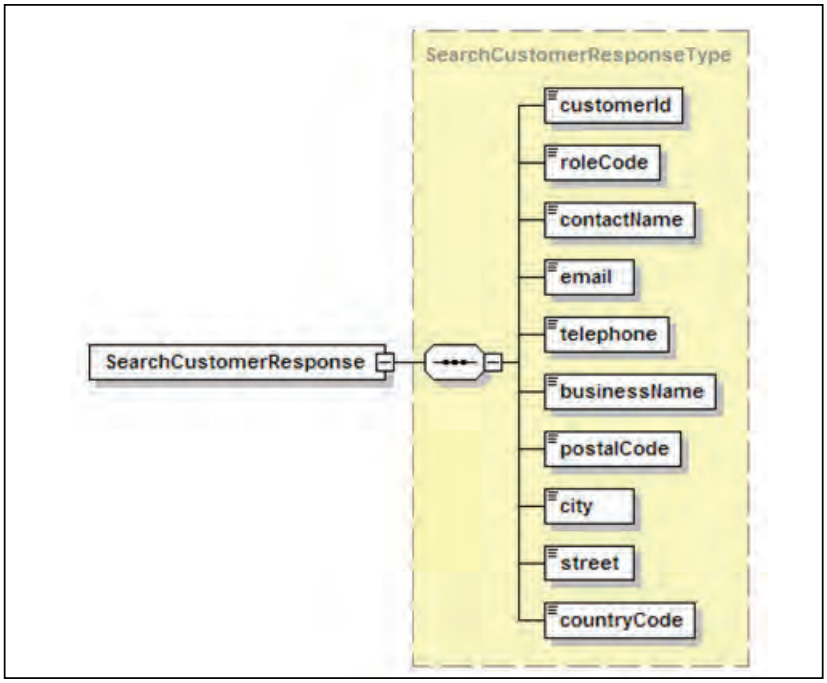


Figure B-188
XSD of message M_4 Search customer Response

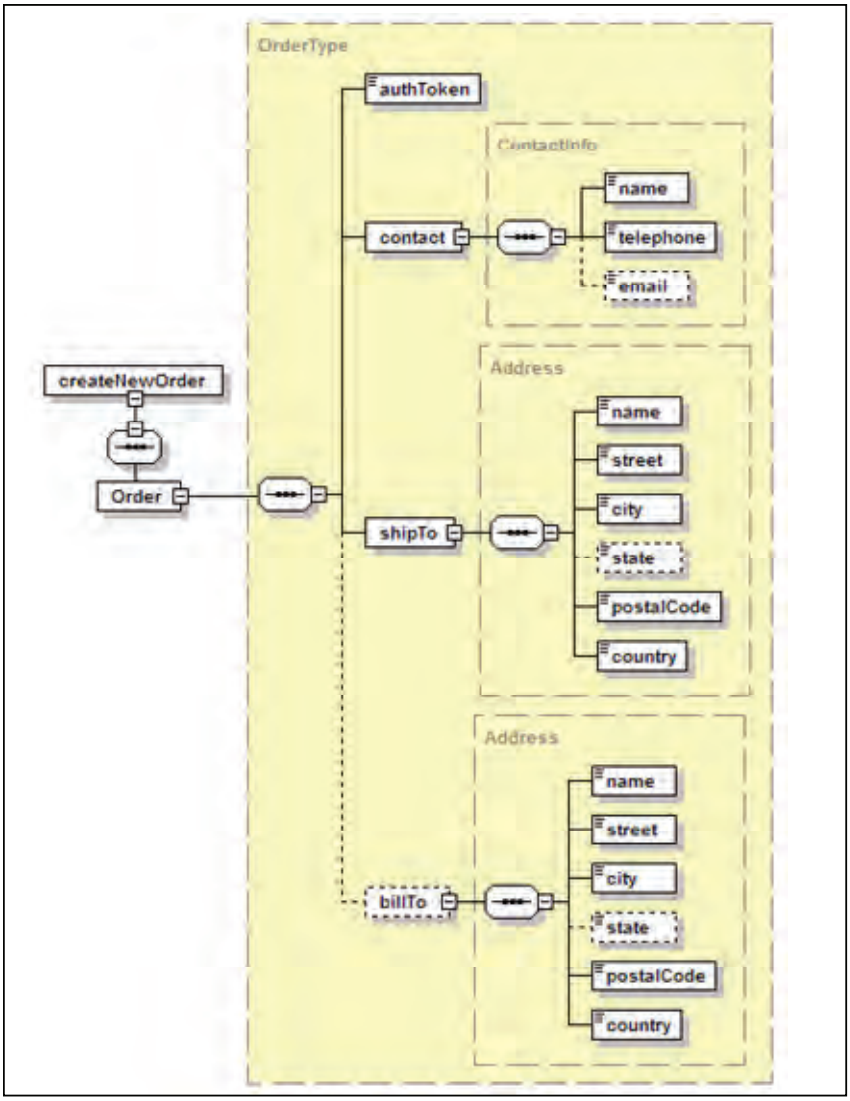


Figure B-189
XSD of message M_5 Create new order Request

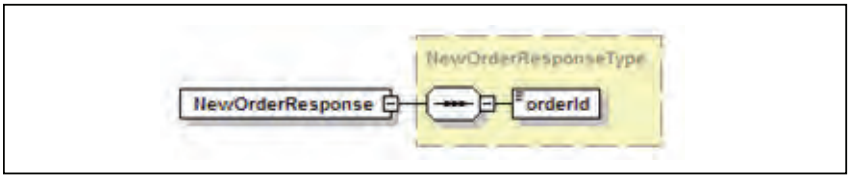


Figure B-190
XSD of message M_6 Create new order Response

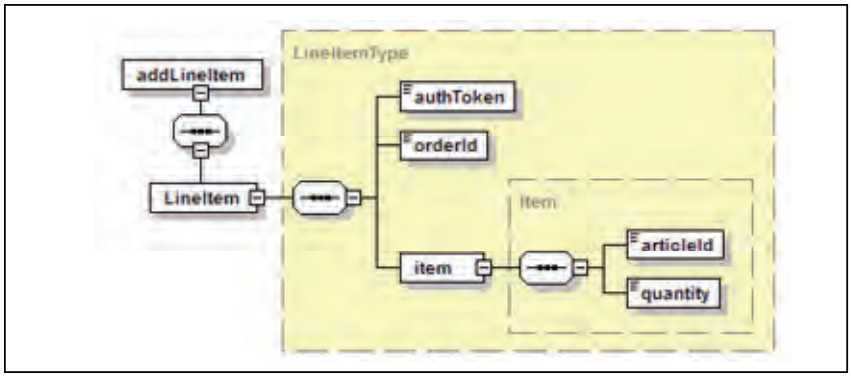


Figure B-191
XSD of message M_7 Add line item Request

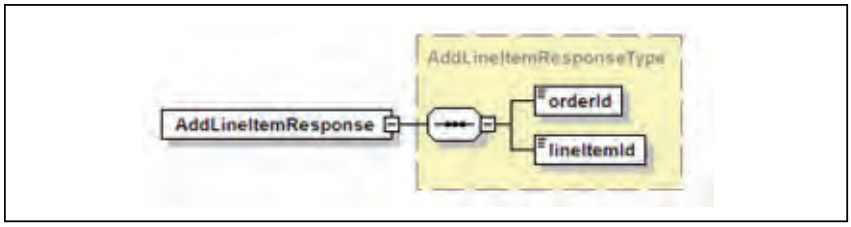


Figure B-192
XSD of message M_8 Add line item Response

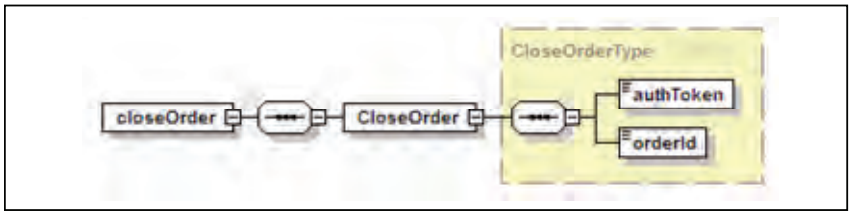


Figure B-193
XSD of message M_9 Close order

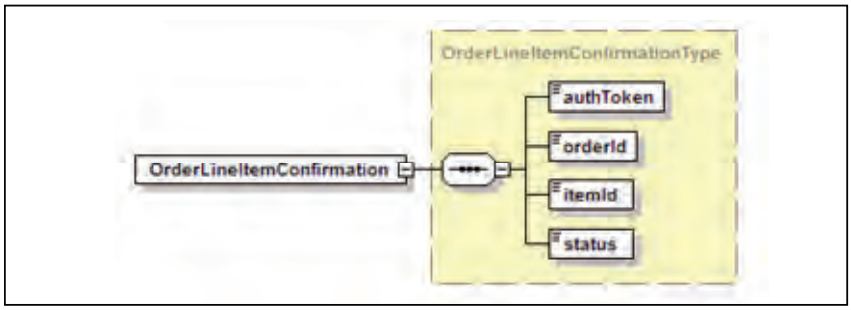


Figure B-194
XSD of message M_{10} Confirm line item

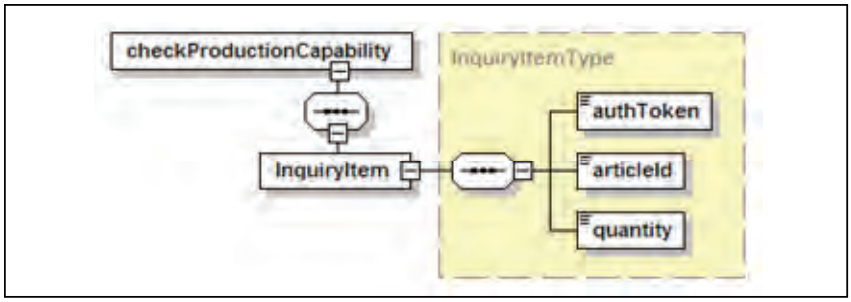


Figure B-195
XSD of message M_{11} Check production capability Request

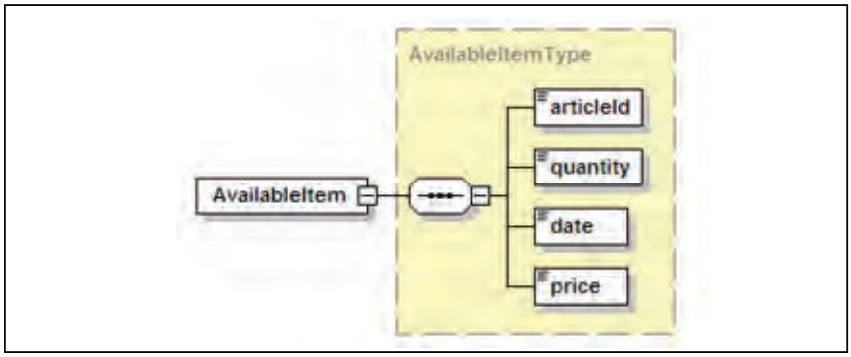


Figure B-196
XSD of message M_{12} Check production capability Response

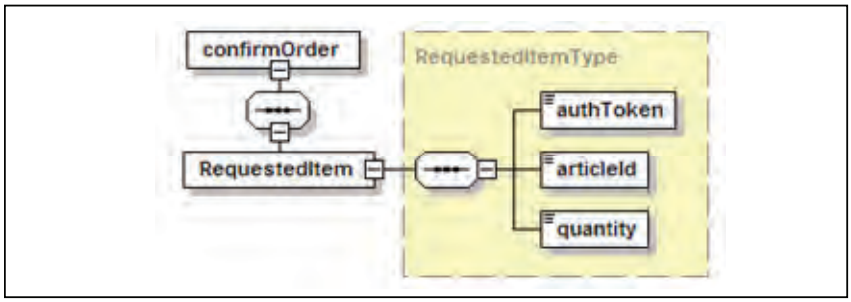


Figure B-197
XSD of message M_{13} Confirm order Request

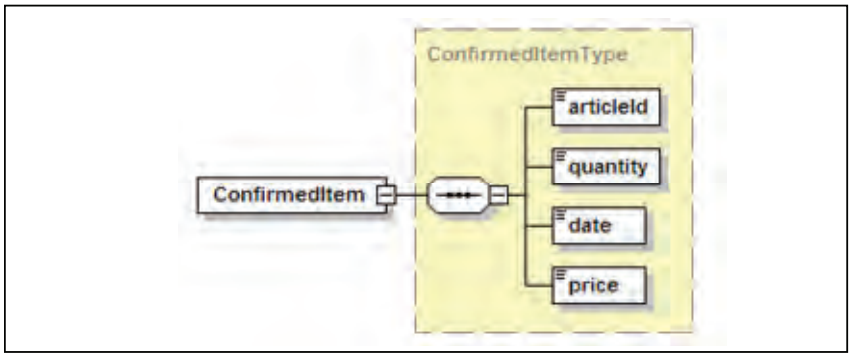


Figure B-198
XSD of message M_{14} Confirm order Response

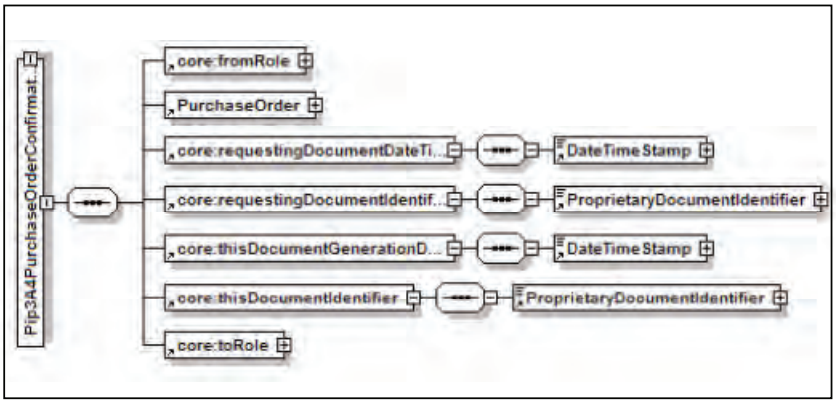


Figure B-199
XSD of message M_{15} PIP 3A4 Purchase Order Confirmation

Figure B-200
XSD of message M_{16} PIP 3A4 Purchase Order Confirmation

Appendix C

The Information Models of Real-Road Operator Case

Figure C-201
The information model of DRC

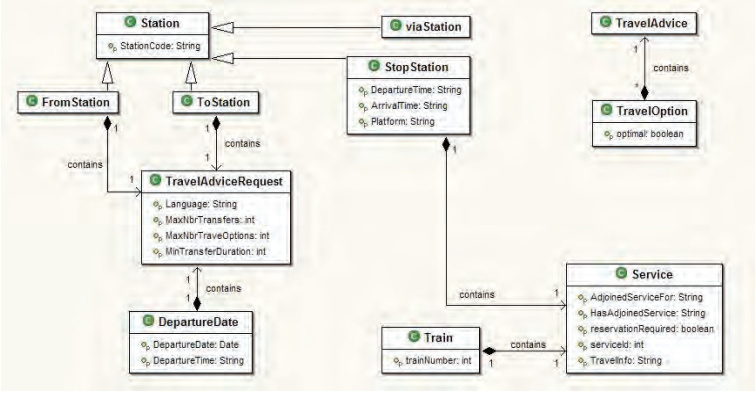


Figure C-202
The information model of CRIS

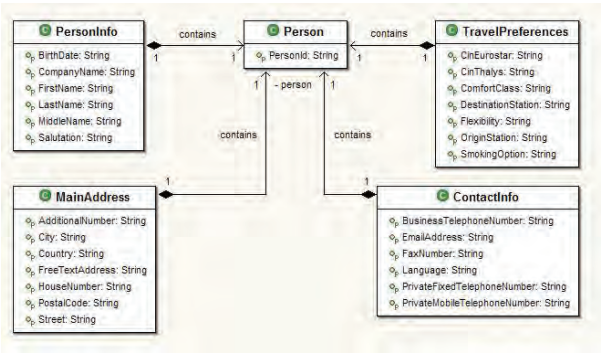


Figure C-203
The information model of TACO

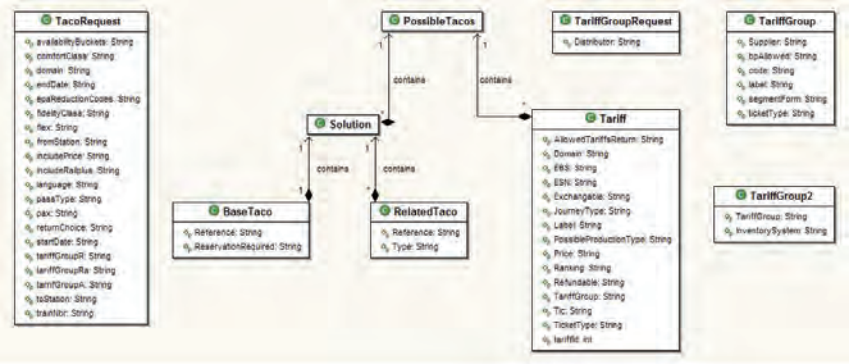
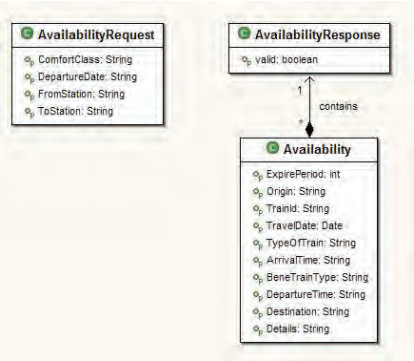


Figure C-204
The information model of TOP100



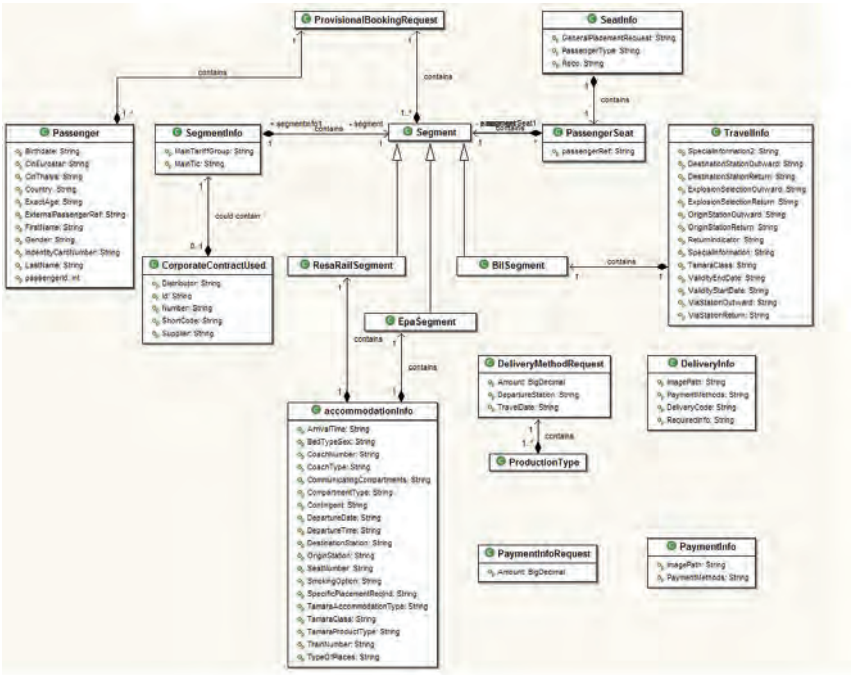


Figure C-205
The information model of ITTI

References

Andrieux, A., Czajkowski, K., Dan, A., Keahey, K., Ludwig, H., Nakata, T., Pruyne, J., Rofrano, J., Tuecke, S., & Xu, M. Web services agreement specification (ws-agreement), 2005.

Baida, Z., Gordijn J. and Omelayenko B. A shared service terminology for online service provisioning. Proceedings of the 6th Int. Conference on Electronic Commerce, vol. 60, 2004, pp. 1-10.

Barnickel, N., Weinand, R. and Flügge, M. Semantic System Integration - Incorporating Rule based Semantic Bridges into BPEL Processes, In: Proceedings of the 6th International Workshop on Evaluation of Ontology-based Tools and the Semantic Web Service Challenge (EON-SWSC-2008), Tenerife, Spain, June 2008.

Battle, S., Bernstein, A., Boley, H., Grosof, B., Gruninger, M., Hull, R., Kifer, M., Martin, D., McIlraith, S., McGuinness, D., Su, J., Tabet, S. Semantic Web Services Framework (SWSF) Overview, W3C Member Submission 9 September 2005, http://www.w3.org/Submission/SWSF/

Beckett, D. Turtle - Terse RDF Triple Language, November 2007, http://www.dajobe.org/2004/01/turtle/

Berners-Lee, T. The World Wide Web and the "Web of Life". World Wide Web Consortium (W3C), 1998. vaialable at http://www.w3.org/People/Berners-Lee/UU.html

Booth, D. (eds.), Haas, H. (eds.), McCabe, F. (eds.), Newcomer, E. (eds.), Champion, M. (eds.), Ferris, C. (eds.) and Orchard D. (eds.). Web Services Architecture. W3C Working Group Note 11 February 2004. http://www.w3.org/TR/ws-arch/

Borst, W.N. Construction of Engineering Ontologies. Centre for Telematica and Information Technology, University of Tweenty. Enshede, The Netherlands, 2007

Bussler, C. B2B Integration/ Concepts and Architecture. Springer-Verlag, 2003. ISBN: 3540434879

Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. The Description Logic Handbook: Theory, Implementation and Applications, Cambridge University Press, 2003. ISBN 0521781760

Chen, P.P.-S.S. The Entity-Relationship Model: Toward a Unified View of Data. ACM Transactions on Database Systems 1(1) pp. 9-36, 1976

Curbera, F., Duftler, M. J., Khalaf, R., Nagy, W. A., Mukhi, N. and Weerawarana, S. Colombo: lightweight middleware for service-oriented computing. IBM Systems Journal 44(4) pp. 799 – 820, 2005

de Bruijn, J., Bussler, C., Domingue, J., Fensel, D., Hepp, M., Keller, U., Kifer, M., König-Ries, B., Kopecky, J., Lara, R., Lausen, H., Oren, E., Polleres, A., Roman, D., Scicluna, J., Stollberg, M. Web Service Modeling Ontology (WSMO), W3C Member Submission 3 June 2005, http://www.w3.org/Submission/WSMO/

de Saussure, F. Course in General Linguistics., Roy Harris (trans.), Open Court Publishing Company, 1986 (original from 1916).

Dean, M. (eds.), Schreiber, G.(eds.), Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., Stein, L.A. OWL Web Ontology Language Reference, W3C Recommendation 10 February 2004, http://www.w3.org/TR/owl-ref/

Department of Defence (DoD), Dictionary of Military Terms, 2001, http://www.dtic.mil/doctrine/jel/doddict/

Dijkman, R.M. Consistency in multi-viewpoint architectural design. PhD thesis, CTIT Ph.D.-thesis series No. 06-80. ISBN 90 75176 80 5, 2006

Dirgahayu, T. and Quartel, D.A.C. and van Sinderen, M.J. (2007) Development of transformations from business process models to implementations by reuse. In: Proceedings of the 3rd International Workshop on Model-Driven Enterprise Information Systems, MDEIS 2007, 12 June 2007, Funchal, Portugal. pp. 41-50. INSTICC Press. ISBN 978-989-8111-00-5

Elmasri, R. and Navathe, S.B. Fundamentals of Database Systems. Benjamin/Cummings, 1994. ISBN 0-8053-1748-1

Erl, T. Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall/PearsonPTR, 2005. ISBN: 0131858580

Estefan, J.A.(eds.), Laskey, K.(eds.), McCabe, F.G.(eds.) and Thornton, D.(eds.). Reference Architecture for Service Oriented Architecture Version 1.0, OASIS Public Review Draft, 23 April 2008, http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra-pr-01.html

European Commission, Interchange of Data between Administrations (IDA) Community Programme. European Interoperability Framework for Pan-European E-Government Services, 2005

Ferreira Pires, L. Architecture Notes: a Framework for Distributed Systems Development. PhD thesis. CTIT Ph. D.-thesis series No. 94-01. ISBN 90-9007461-9, 1994

García, R. and Celma, Ò. Semantic Integration and Retrieval of Multimedia Metadata. 5th International Workshop on Knowledge Markup and Semantic Annotation, SemAnnot 2005, 7th November 2005, Galway, Ireland. http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-185/

Goh, C.H. Representing and Reasoning about Semantic Conflicts in Heterogeneous Information Sources. PhD thesis, MIT, 1997.

Gordijn, J. and Akkermans, H. E3-value: Design and evaluation of e-business models. IEEE Intelligent Systems, 16(4):11–17, 2001

Gruber, T.R. A translation approach to portable ontologies. Knowledge Acquisition 5 (2) pp. 199-220, 1993.

Guarino, N. Formal Ontology in Information Systems. In N. Guarino (ed.) Formal Ontology in Information Systems. Proceedings of FOIS'98, Trento, Italy, 6-8 June 1998. IOS Press, Amsterdam: 3-15.

Haase, P. and Motik, B. A mapping system for the integration of OWL-DL ontologies. In Proceedings of the First international Workshop on interoperability of Heterogeneous information Systems (Bremen, Germany, November 04 - 04, 2005). IHIS '05. ACM, New York, NY, 9-16.

IBM. IBM service definition, http://www.research.ibm.com/ssme/services.shtml, 2006.

Institute of Electrical and Electronics Engineers (IEEE). IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries, 1990

International Organization for Standardization (ISO), DIS 16100: Manufacturing Software Capability Profiling, Part 1 - Framework for interoperability, 2000.

International Organization for Standardization (ISO), ISO/IEC 2382-1:1993 Information technology -- Vocabulary -- Part 1: Fundamental terms, 1993

Janssen, W., Jonkers, H. and Verhoosel, J. What Makes Business Processes Special? An evaluation framework for modeling languages and tools in Business Process Redesign, In Siau, Wand and Parsons (Eds.), Proceedings 2nd CAiSE/IFIP 8.1 international workshop on evaluation of modeling methods in systems analysis and design, Barcelona, June, 1997

Jensen, K. Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts. Monographs in Theoretical Computer Science, Springer-Verlag, 1992. ISBN: 3-540-60943-1.

Jensen, K. Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods. Monographs in Theoretical Computer Science, Springer -Verlag, 1994. ISBN: 3 -540-58276-2

Jonkers, H., Lankhorst, M., van Buuren, R., Hoppenbrouwers, S., Bonsangue, M., and van der Torre, L. Concepts for Modelling Enterprise Architectures. International Journal of Cooperative Information Systems, vol. 13, no. 3, 2004, pp. 257-287.

Klein, M. Combining and Relating Ontologies: an Analysis of Problems and Solutions. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01), Workshop on

Ontologies and Information Sharing, Vol. 47, pp. 53–62, Seattle, USA, August 2001.

Koehoorn, B. Comparing Systems for Replacement: A Constraint-based Approach for Comparing ISDL Models. M.Sc. Thesis Business Information Technology, University of Twente. March 2007.

Margaria T., Bakera M., Raffelt H. and Steffen B. Synthesizing the mediator with jabc/abc, In: Proceedings of the 6th International Workshop on Evaluation of Ontology-based Tools and the Semantic Web Service Challenge (EON-SWSC-2008), Tenerife, Spain, June 2008.

Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N., Sycara, K. OWL-S: Semantic Markup for Web Services W3C Member Submission 22 November 2004, http://www.w3.org/Submission/OWL-S/

Morris, C.W. Foundations of the Theory of Signs. In: International Encyclopaedia of Unified Science (Neurath, O., Carnap, R., Morris, C. (eds), Chicago University Press, Chicago, pp. 77-138, 1938

Naiman, C.F. and Ouksel, A. M. A classification of semantic conflicts in heterogeneous database systems. J. of Organizational Computing 5(2): pp. 167-193 (1995)

Ogden, C.K., Richards, I.A. The Meaning of Meaning: A Study of the Influence of Language Upon Thought and of the Science of Symbolism., New York: Harcourt Brace Jovanovich, 1923.

Pollock, J.T. and Hodgson, R. Adaptive Information: Improving Business Through Semantic Interoperability, Grid Computing, and Enterprise Integration, Wiley-Interscience, ISBN: 0471488542, 2004

Quartel, D.A.C., Dijkman R. and van Sinderen M. Methodological support for service-oriented design with ISDL. Proceedings of the 2nd Internatiation Conference on Service Oriented Computing, 2004, pp. 1-10.

Quartel, D.A.C., Dijkman R.M., Sinderen van M.J. Methodological support for service-oriented design with ISDL. In: Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC 2004), New York City, NY, USA, 2004.

Quartel, D.A.C., Ferreira Pires L., van Sinderen M.J., Franken H. and Vissers C. On the role of basic design concepts in behaviour structuring. Computer Networks and ISDN Systems, 29:413–436, 1997.

Quartel, D.A.C., Ferreira Pires, L., Sinderen, van M.J. On Architectural Support for Behaviour Refinement in Distributed Systems Design. In: Journal of integrated design and process science online, 06(01) ISNN 1092-0617.

Quartel, D.A.C., Steen, M.W.A., Pokraev, S.V. and van Sinderen, M.J. COSMO: a conceptual framework for service modelling and refinement. Information Systems Frontiers, 9 (2-3). pp. 225-244. ISSN 1387-3326, 2007

Quartel, D.A.C.. Action relations: Basic Design Concepts for Behaviour Modelling and Refinement. PhD thesis, CTIT Ph.D.-thesis series No. 98-18. ISBN 90 365 1071 6, 1997

Quartel, D.A.C.. Simulation and execution of service models using ISDL. In: Proceedings of the 1st International Workshop on Architectures, Concepts and Technologies for Service-Oriented Computing, ACT4SOC 2007, July 22, 2007, Barcelona, Spain. INSTICC Press, pp. 19-27. ISBN 978-989-8111-08-1.

Ratzer, A.V., Wells, L., Lassen, H.M., Laursen, M., Qvortrup, J.F., Stissing, M.S., Westergaard, M., Christensen, S., Jensen, K. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Net, In: Proceedings of the 24th International Conference on Applications and Theory of Petri Nets (ICATPN 2003), Eindhoven, The Netherlands, June 23-27, 2003, pages 450-462. Volume 2679 of Lecture Notes in Computer Science / Wil M.P. van der Aalst and Eike Best (Eds.) Springer-Verlag, June 2003.

Sheth, A.P. and Kashyap, V. So Far (Schematically) yet So Near (Semantically). In: Hsiao D. K., Neuhold, E.J. and Sacks-Davis, R. (eds). Proceedings of the IFIP WG 2.6 Database Semantics Conference on Interoperable Database Systems (DS-5), Lorne, Victoria, Australia, pp. 283-312 (1992)

Sheth, A.P. and Larson, J. Federated database systems for managing distributed, heterogeneous, and autonomous databases ACM Computing Surveys, 1990.

Shvaiko, P. and Euzenat, J.A survey of schema-based matching approaches. Journal on Data Semantics, 4:146–171, 2005

Siau, K. and Rossi, M. Evaluation of Information Modeling Methods - A Review. HICSS (5). pp. 314-322, 1998.

Sprott D.. and Wilkes L. Understanding Service-Oriented Architecture. CBDI Journal, CBDI Forum, January 2004.

Steffen B., Margaria T., Nagel R., Jörges S., and Kubczak C. Model-Driven Development with the jABC. In: Proceedings of Haifa Verification Conference, LNCS N.4383. Springer Verlag, 2006.

Studer, R., Benjamins, V.R. and Fensel, D. Knowledge engineering: principles and methods. IEEE Transactions on Data and Knowledge Engineering 25(1-2):161-197, 1998

SWSC, Purchase Order Mediation Scenario, 2007, http://sws-challenge.org/wiki/index.php/Scenario:Purchase Order Mediation v2

U.S. Department of Commerce (DOC), National Telecommunications and Information Administration (NTIA). Telecommunications: Glossary of Telecommunication Terms, 1996

Ullmann, S. .Semantics: An Introduction to the Science of Meaning., Basil Blackwell, Oxford,1972.

van Eck, P., Blanken, H. and Wieringa, R.J. Project GRAAL: Towards Operational Architecture Alignment. International Journal of Cooperative Information Systems 13(3), 2004, pp. 235-255.

van Sinderen, M.J. van, Ferreira Pires, L., Vissers, C. A., Katoen, J.P. A design model for open distributed processing systems. Computer Networks and ISDN Systems, Vol. 27, 1995, pp. 1263-1285. ISSN 0169-7552.

Verma, K., Gomadam, K., Sheth, A., Miller, J., Wu, Z. The METEOR-S Approach for Configuring and Executing Dynamic Web Processes", Technical Report . Date: 6-24-05.

Visser, P.R.S., Jones, D. M., Bench-Capon, T.J.M. and Shave, M.J.R. An Analysis of Ontology Mismatches; Heterogeneity versus Interoperability", American Association for Artificial Intelligence (AAAI 1997) Spring Symposium on Ontological Engineering, pp. 164–172, Stanford University, California, USA. 1997

Vissers C, Logrippo L. The importance of the service concept in the design of data communication protocols. Protocol Specification, Testing and Verification, V, 1986, pp. 3-17.

Wieringa R.J. Design Methods for Reactive Systems: Yourdon, Statemate, and the UML. Design Methods for Reactive Systems: Yourdon, Statemate, and the UML. Morgan Kaufmann, 2003. http://www.mkp.com/dmrs

Wieringa, R.J., Maiden, N.A.M., Mead, N.R. and Rolland, C. Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. In: Requirements Engineering 11 (1) pp. 102-107, 2006

Wood, J. What's in a link? Readings in Knowledge Representation, Morgan Kaufmann, 1985.

Summary

In this thesis, we propose *a method for the semantic integration of service oriented applications.* The distinctive feature of the method is that *semantically-enriched service models* are employed at *different levels of abstraction* (from business requirements to software implementation) to deliver *flexible integration solutions*.

In Chapter 2, we start with analyzing the most cited *interoperability* definitions and derive common characteristics of interoperability. Next, we use these common characteristics to define what interoperability means and identify three different levels of interoperability, namely, *syntactic*, *semantic* and *pragmatic interoperability*. Finally, we study literature from different areas and identify possible *interoperability problems* at each of the interoperability levels.

In Chapter 3, we present a short history of the enterprise application integration (EAI) approaches, discuss their shortcomings and argue what is required to address these shortcomings. We identify three main aspects of the EAI problem. The first aspect concerns the difference in the *information models* of the systems that have to be integrated. The second aspect concerns the differences in the *interaction protocols* of the systems. Finally, the third aspect concerns the *complexity* of building EAI solutions.

Service-Oriented Architecture (SOA), Knowledge Representation (KR) and Model-Driven Architectures (MDA) have been proposed as solutions to each of the indentified problems. In Chapter 3, we argue that, since the problem aspects of current EAI approaches always occur together, SOA, KR and MDA should be combined to deal with the problem as a whole.

In Chapter 4, we define a *conceptual framework for service modeling*. The purpose of the framework is to serve as a *common semantic meta-model* that enables the description, integration and reasoning about (integrated) service-oriented applications. Using the framework one can model the domain of a system, the interactions among its components and their relations, and reason whether these components are interoperable. We expect that our framework will have a wide spectrum of application, e.g., can be used to model services at a business, application and component level, thus beyond the usual domain of web services.

In Chapter 5, we present a *method for the semantic integration of service-oriented applications*. We start by identifying *necessary conditions* for semantic

and pragmatic interoperability of service-oriented applications. Next, we propose an *integration method* that enables business domain experts to explicitly specify an integration solution at a higher level of abstraction. The abstract solution is then (semi-)automatically transformed to a software solution by adding technical details by the IT experts. Finally, we present a method to *verify formally* whether the proposed integration solution meets the identified conditions for interoperability.

In Chapters 6 to 9, we validate our integration method by applying it a particular context, using particular technologies. In Chapter 1, we identified a number of *requirements* for integration methods in general. To verify whether our method meets these requirements we make a number of *claims* and provide *arguments for their validity*. We do this by applying our method to in a concrete context using concrete technologies. For that purpose, we solve two integration problems from order management domain and travel domain, respectively. When applying our integration method we observe a number of effects. We analyse our observations and argue to what extent our integration method meets the requirements defined in Chapter 1.

Finally, in Chapter 10, we summarise the conclusions of this thesis and identify some topics for further research.

Publications by the Author

(listed in reverse chronological order):

Quartel, D.A.C., Pokraev, S.V., Dirgahayu, T., Mantovaneli Pessoa, R. and van Sinderen, M.J. Model-driven service integration using the COSMO framework. In: Semantic Web Services Challenge: Proceedings of the 2008 Workshops, 26 Oct 2008, Karlsruhe, Germany. pp. 77-88. Stanford Logic Group Technical Reports (LG-2009-01)

Quartel, D.A.C., Pokraev, S.V., Mantovaneli Pessoa, R. and van Sinderen, M.J. Model-driven development of a mediation service. In: Twelfth International IEEE Enterprise Computing Conference, EDOC 2008, 15-19 Sep 2008, Munich, Germany. pp. 117-126. IEEE Computer Society Press. ISSN 1541-7719 ISBN 978-0-7695-3373-5

Pokraev, S.V., Quartel, D.A.C., Steen, M.W.A., Wombacher, A. and Reichert, M.U. (2007) Business Level Service-Oriented Enterprise Application Integration. In: Proceedings 3rd International Conference on Interoperability for Enterprise Software and Applications (I-ESA 2007), 28 - 30 Mar 2007, Funchal (Madeira Island), Portugal. pp. 507-518. Springer Verlag. ISBN 978-1-84628-857-9

Quartel, D.A.C., Steen, M.W.A., Pokraev, S.V., van Sinderen, M.J. COSMO: a conceptual framework for service modelling and refinement. Information Systems Frontiers, 9 (2-3). pp. 225-244. ISSN 1387-3326

Pokraev, S.V., Quartel, D.A.C., Steen, M.W.A. and Reichert, M.U. Requirements and Method for Assessment of Service Interoperability. In: Proceedings of the Fourth International Conference on Service Oriented Computing (ICSOC'06), 4-7 Dec 2006, Chicago. pp. 1-14. Lecture Notes in Computer Science 4294. Springer Verlag. ISSN 0302-9743 ISBN 978-3-540-68147-2

Pokraev, S.V., Quartel, D.A.C., Steen, M.W.A. and Reichert, M.U. Semantic Service Modeling - Enabling System Interoperability. In: Proc. International Conference on Interoperability for Enterprise Software and Applications (I-ESA'06), Mar 2006, Bordeaux, France. pp. 221-231. Springer Verlag. ISBN 978-1-84628-713-8

Pokraev, S.V., Quartel, D.A.C., Steen, M.W.A. and Reichert, M.U. (2006) A Method for Formal Verification of Service Interoperability. In: Proceedings IEEE International Conference on Web Services (ICWS'06), 18 - 22 September 2006, Chicago, USA. pp. 895-900. IEEE Computer Society. ISBN 0-7695-2669-1

Pokraev, S.V., Reichert, M.U. Mediation Patterns for Message Exchange Protocols. In: Proceedings of the CAiSE'06 Workshops / Open INTEROP Workshop on Enterprise Modelling and Ontologies for Interoperability (EMOI-INTEROP), 5 - 9 June 2006, Luxembourg. pp. 659-663. Presses Universitaires de Namur. ISBN 2-87037-525-5

Quartel, D.A.C., Steen, M.W.A., Pokraev, S.V. and van Sinderen, M.J. A Conceptual Framework for Service Modelling. In: Proceedings Tenth IEEE International EDOC Enterprise Computing Conference, 16-20 Oct 2006, Hong Kong. pp. 319-330. IEEE Computer Society Press. ISSN 1541-7719 ISBN 978-0-7695-2558-7

Diakov, N.K., Zlatev, Z.V. and Pokraev, S.V. Composition of Negotiation Protocols for E-Commerce Applications. In: IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05), 29 Mar-1 Apr 2005, Hong Kong, China. pp. 418-423. IEEE Computer Science Press. ISBN 0769520731

Pokraev, S.V., Koolwaaij, J., van Setten, M., Broens, T.H.F., Dockhorn Costa, P., Wibbels, M., Ebben, P. and Strating, P. Service platform for rapid development and deployment of context-Aware, mobile applications. In: International Conference on Webservices (ICWS'05), Industry track, 2005, Orlando, Florida, USA. pp. 639-646. IEEE Computer Society Press. ISBN 0-7695-2409-5

Pokraev, S.V., Reichert, M.U., Steen, M.W.A. and Wieringa, R.J. Semantic and Pragmatic Interoperability: A Model for Understanding. In: Proceedings of the Open Interop Workshop on Enterprise Modelling and Ontologies for Interoperability (EMOI - INTEROP'05), 13 - 14 Jun 2005, Porto, Portugal. pp. 1-5. CEUR Workshop Proceedings 160. CEUR-WS.org. ISSN 1613-0073

Broens, T.H.F., Pokraev, S.V., van Sinderen, M.J., Koolwaaij, J. and Dockhorn Costa, P. Context-aware, ontology-based, service discovery. In: European Symposium on Ambient Intelligence (EUSAI), Eindhoven, The Netherlands. pp. 72-83. Lecture Notes in Computer Science 3295. Springer. ISBN 3540237216

Pokraev, S.V., Wieringa, R.J. and Steen, M.W.A. Towards semantic service specification and discovery. In: CAiSE'04 Workshops in connection with The 16th Conference on Advanced Information Systems Engineering, 7-11 June 2004, Riga, Latvia. pp. 363-367. Faculty of Computer Science and Information Technology. ISBN 9984976734

Pokraev, S.V., Zlatev, Z.V., Brussee, R. and van Eck, P.A.T. Semantic Support for Automated Negotiation with Alliances. In: 6th International conference on enterprise information systems (ICEIS 2004), 14-17 April 2004, Porto, Portugal. pp. 244-249. INSTICC.

Zlatev, Z.V., Diakov, N.K. and Pokraev, S.V. Construction of Negotiation Protocols for E-commerce Applications. Technical Report SEN-R0417 Center for Mathematics and Computer Science, Amsterdam. ISSN 1381-3625

Zlatev, Z.V. and Diakov, N.K. and Pokraev, S.V. (2004) Construction of Negotiation Protocols for E-Commerce Applications. ACM SIGecom Exchanges, 5 (2). pp. 12-22. ISSN 1551-9031

Pokraev, S.V., Koolwaaij, J. and Wibbels, M. Extending UDDI with Context-Aware Features Based on Semantic Service Descriptions. In: The International Conference on Web Services (ICWS '03), 23-26 June 2003, Las Vegas, Nevada, USA. pp. 184-190. CSREA Press. ISBN 1892512491

Michiels, E.F., Widya, I.A, Volman, C.J.A.M., Pokraev, S.V. and de Diana, I.P.F. On the Enterprise Modelling of an Educational Information Infrastructure. In: Enterprise Information Systems III, Setubal, Portugal. pp. 231-239. Kluwer Academic Publishers. ISBN 1-4020-0563-6

Widya, I.A., Volman, C.J.A.M., Pokraev, S.V., de Diana, I.P.F. and Michiels, E.F. Enterprise Modelling for an Educational Information Infrastructure. In: The 3rd International Conference on Enterprise Information Systems, 2001, Setubal, Portugal. pp. 785-792. ICEIS Press. ISBN 972-98050-2-4

Michiels, E.F., Widya, I.A., Volman, C.J.A.M., Pokraev, S.V. and de Diana, I.P.F. On the Enterprise Modelling of an Educational Information Infrastructure. Technical Report TR-CTIT-00-18, Centre for Telematics and Information Technology, University of Twente, Enschede. ISSN 1381-3625

SIKS Dissertation Series

1998

[1998-1] Johan van den Akker (CWI) DEGAS - An Active, Temporal Database of Autonomous Objects

[1998-2] Floris Wiesman (UM) Information Retrieval by Graphically Browsing Meta-Information

[1998-3] Ans Steuten (TUD) A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective

[1998-4] Dennis Breuker (UM) Memory versus Search in Games

[1998-5] E.W.Oskamp (RUL) Computerondersteuning bij Straftoemeting

1999

[1999-1] Mark Sloof (VU) Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products

[1999-2] Rob Potharst (EUR) Classification using decision trees and neural nets

[1999-3] Don Beal (UM) The Nature of Minimax Search

[1999-4] Jacques Penders (UM) The practical Art of Moving Physical Objects

[1999-5] Aldo de Moor (KUB) Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems

[1999-6] Niek J.E. Wijngaards (VU) Re-design of compositional systems

[1999-7] David Spelt (UT) Verification support for object database design

[1999-8] Jacques H.J. Lenting (UM) Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

2000

[2000-1] Frank Niessink (VU) Perspectives on Improving Software Maintenance

[2000-2] Koen Holtman (TUE) Prototyping of CMS Storage Management

[2000-3] Carolien M.T. Metselaar (UVA) Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.

[2000-4] Geert de Haan (VU) ETAG, A Formal Model of Competence Knowledge for User Interface Design

[2000-5] Ruud van der Pol (UM) Knowledge-based Query Formulation in Information Retrieval.

[2000-6] Rogier van Eijk (UU) Programming Languages for Agent Communication

[2000-7] Niels Peek (UU) Decision-theoretic Planning of Clinical Patient Management

[2000-8] Veerle Coup, (EUR) Sensitivity Analyis of Decision-Theoretic Networks
[2000-9] Florian Waas (CWI) Principles of Probabilistic Query Optimization
[2000-10] Niels Nes (CWI) Image Database Management System Design Considerations, Algorithms and Architecture
[2000-11] Jonas Karlsson (CWI) Scalable Distributed Data Structures for Database Management

2001
[2001-1] Silja Renooij (UU) Qualitative Approaches to Quantifying Probabilistic Networks
[2001-2] Koen Hindriks (UU) Agent Programming Languages: Programming with Mental Models
[2001-3] Maarten van Someren (UvA) Learning as problem solving
[2001-4] Evgueni Smirnov (UM) Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets
[2001-5] Jacco van Ossenbruggen (VU) Processing Structured Hypermedia: A Matter of Style
[2001-6] Martijn van Welie (VU) Task-based User Interface Design
[2001-7] Bastiaan Schonhage (VU) Diva: Architectural Perspectives on Information Visualization
[2001-8] Pascal van Eck (VU) A Compositional Semantic Structure for Multi-Agent Systems Dynamics.
[2001-9] Pieter Jan 't Hoen (RUL) Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes
[2001-10] Maarten Sierhuis (UvA) Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design
[2001-11] Tom M. van Engers (VUA) Knowledge Management: The Role of Mental Models in Business Systems Design

2002
[2002-01] Nico Lassing (VU) Architecture-Level Modifiability Analysis
[2002-02] Roelof van Zwol (UT) Modelling and searching web-based document collections
[2002-03] Henk Ernst Blok (UT) Database Optimization Aspects for Information Retrieval
[2002-04] Juan Roberto Castelo Valdueza (UU) The Discrete Acyclic Digraph Markov Model in Data Mining
[2002-05] Radu Serban (VU) The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents
[2002-06] Laurens Mommers (UL) Applied legal epistemology; Building a knowledge-based ontology of the legal domain

[2002-07] Peter Boncz (CWI) Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications
[2002-08] Jaap Gordijn (VU) Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas
[2002-09] Willem-Jan van den Heuvel(KUB) Integrating Modern Business Applications with Objectified Legacy Systems
[2002-10] Brian Sheppard (UM) Towards Perfect Play of Scrabble
[2002-11] Wouter C.A. Wijngaards (VU) Agent Based Modelling of Dynamics: Biological and Organisational Applications
[2002-12] Albrecht Schmidt (Uva) Processing XML in Database Systems
[2002-13] Hongjing Wu (TUE)A Reference Architecture for Adaptive Hypermedia Applications
[2002-14] Wieke de Vries (UU) Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems
[2002-15] Rik Eshuis (UT) Semantics and Verification of UML Activity Diagrams for Workflow Modelling
[2002-16] Pieter van Langen (VU) The Anatomy of Design: Foundations, Models and Applications
[2002-17] Stefan Manegold (UVA) Understanding, Modeling, and Improving Main-Memory Database Performance

2003

[2003-01] Heiner Stuckenschmidt (VU) Ontology-Based Information Sharing in Weakly Structured Environments
[2003-02] Jan Broersen (VU) Modal Action Logics for Reasoning About Reactive Systems
[2003-03] Martijn Schuemie (TUD) Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy
[2003-04] Milan Petkovic (UT) Content-Based Video Retrieval Supported by Database Technology
[2003-05] Jos Lehmann (UVA) Causation in Artificial Intelligence and Law - A modelling approach
[2003-06] Boris van Schooten (UT) Development and specification of virtual environments
[2003-07] Machiel Jansen (UvA) Formal Explorations of Knowledge Intensive Tasks
[2003-08] Yongping Ran (UM) Repair Based Scheduling
[2003-09] Rens Kortmann (UM) The resolution of visually guided behaviour
[2003-10] Andreas Lincke (UvT) Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture
[2003-11] Simon Keizer (UT) Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

[2003-12] Roeland Ordelman (UT) Dutch speech recognition in multimedia information retrieval
[2003-13] Jeroen Donkers (UM) Nosce Hostem - Searching with Opponent Models
[2003-14] Stijn Hoppenbrouwers (KUN) Freezing Language: Conceptualisation Processes across ICT-Supported Organisations
[2003-15] Mathijs de Weerdt (TUD) Plan Merging in Multi-Agent Systems
[2003-16] Menzo Windhouwer (CWI) Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses
[2003-17] David Jansen (UT) Extensions of Statecharts with Probability, Time, and Stochastic Timing
[2003-18] Levente Kocsis (UM) Learning Search Decisions

2004
[2004-01] Virginia Dignum (UU) A Model for Organizational Interaction: Based on Agents, Founded in Logic
[2004-02] Lai Xu (UvT) Monitoring Multi-party Contracts for E-business
[2004-03] Perry Groot (VU) A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving
[2004-04] Chris van Aart (UVA) Organizational Principles for Multi-Agent Architectures
[2004-05] Viara Popova (EUR) Knowledge discovery and monotonicity
[2004-06] Bart-Jan Hommes (TUD) The Evaluation of Business Process Modeling Techniques
[2004-07] Elise Boltjes (UM) Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes
[2004-08] Joop Verbeek(UM) Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politi%ole gegevensuitwisseling en digitale expertise
[2004-09] Martin Caminada (VU) For the Sake of the Argument; explorations into argument-based reasoning
[2004-10] Suzanne Kabel (UVA) Knowledge-rich indexing of learning-objects
[2004-11] Michel Klein (VU) Change Management for Distributed Ontologies
[2004-12] The Duy Bui (UT) Creating emotions and facial expressions for embodied agents
[2004-13] Wojciech Jamroga (UT) Using Multiple Models of Reality: On Agents who Know how to Play
[2004-14] Paul Harrenstein (UU) Logic in Conflict. Logical Explorations in Strategic Equilibrium
[2004-15] Arno Knobbe (UU) Multi-Relational Data Mining
[2004-16] Federico Divina (VU) Hybrid Genetic Relational Search for Inductive Learning
[2004-17] Mark Winands (UM) Informed Search in Complex Games

[2004-18] Vania Bessa Machado (UvA) Supporting the Construction of Qualitative Knowledge Models
[2004-19] Thijs Westerveld (UT) Using generative probabilistic models for multimedia retrieval
[2004-20] Madelon Evers (Nyenrode) Learning from Design: facilitating multidisciplinary design teams

2005
[2005-01] Floor Verdenius (UVA) Methodological Aspects of Designing Induction-Based Applications
[2005-02] Erik van der Werf (UM)) AI techniques for the game of Go
[2005-03] Franc Grootjen (RUN) A Pragmatic Approach to the Conceptualisation of Language
[2005-04] Nirvana Meratnia (UT) Towards Database Support for Moving Object data
[2005-05] Gabriel Infante-Lopez (UVA) Two-Level Probabilistic Grammars for Natural Language Parsing
[2005-06] Pieter Spronck (UM) Adaptive Game AI
[2005-07] Flavius Frasincar (TUE) Hypermedia Presentation Generation for Semantic Web Information Systems
[2005-08] Richard Vdovjak (TUE) A Model-driven Approach for Building Distributed Ontology-based Web Applications
[2005-09] Jeen Broekstra (VU) Storage, Querying and Inferencing for Semantic Web Languages
[2005-10] Anders Bouwer (UVA) Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments
[2005-11] Elth Ogston (VU) Agent Based Matchmaking and Clustering - A Decentralized Approach to Search
[2005-12] Csaba Boer (EUR) Distributed Simulation in Industry
[2005-13] Fred Hamburg (UL) Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen
[2005-14] Borys Omelayenko (VU) Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics
[2005-15] Tibor Bosse (VU) Analysis of the Dynamics of Cognitive Processes
[2005-16] Joris Graaumans (UU) Usability of XML Query Languages
[2005-17] Boris Shishkov (TUD) Software Specification Based on Re-usable Business Components
[2005-18] Danielle Sent (UU) Test-selection strategies for probabilistic networks
[2005-19] Michel van Dartel (UM) Situated Representation
[2005-20] Cristina Coteanu (UL) Cyber Consumer Law, State of the Art and Perspectives
[2005-21] Wijnand Derks (UT) Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006

[2006-01] Samuil Angelov (TUE) Foundations of B2B Electronic Contracting
[2006-02] Cristina Chisalita (VU) Contextual issues in the design and use of information technology in organizations
[2006-03] Noor Christoph (UVA) The role of metacognitive skills in learning to solve problems
[2006-04] Marta Sabou (VU) Building Web Service Ontologies
[2006-05] Cees Pierik (UU) Validation Techniques for Object-Oriented Proof Outlines
[2006-06] Ziv Baida (VU) Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling
[2006-07] Marko Smiljanic (UT) XML schema matching -- balancing efficiency and effectiveness by means of clustering
[2006-08] Eelco Herder (UT) Forward, Back and Home Again - Analyzing User Behavior on the Web
[2006-09] Mohamed Wahdan (UM) Automatic Formulation of the Auditor's Opinion
[2006-10] Ronny Siebes (VU) Semantic Routing in Peer-to-Peer Systems
[2006-11] Joeri van Ruth (UT) Flattening Queries over Nested Data Types
[2006-12] Bert Bongers (VU) Interactivation - Towards an e-cology of people, our technological environment, and the arts
[2006-13] Henk-Jan Lebbink (UU) Dialogue and Decision Games for Information Exchanging Agents
[2006-14] Johan Hoorn (VU) Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change
[2006-15] Rainer Malik (UU) CONAN: Text Mining in the Biomedical Domain
[2006-16] Carsten Riggelsen (UU) Approximation Methods for Efficient Learning of Bayesian Networks
[2006-17] Stacey Nagata (UU) User Assistance for Multitasking with Interruptions on a Mobile Device
[2006-18] Valentin Zhizhkun (UVA) Graph transformation for Natural Language Processing
[2006-19] Birna van Riemsdijk (UU) Cognitive Agent Programming: A Semantic Approach
[2006-20] Marina Velikova (UvT) Monotone models for prediction in data mining
[2006-21] Bas van Gils (RUN) Aptness on the Web
[2006-22] Paul de Vrieze (RUN) Fundaments of Adaptive Personalisation
[2006-23] Ion Juvina (UU) Development of Cognitive Model for Navigating on the Web
[2006-24] Laura Hollink (VU) Semantic Annotation for Retrieval of Visual Resources

[2006-25] Madalina Drugan (UU) Conditional log-likelihood MDL and Evolutionary MCMC
[2006-26] Vojkan Mihajlovic (UT) Score Region Algebra: A Flexible Framework for Structured Information Retrieval
[2006-27] Stefano Bocconi (CWI) Vox Populi: generating video documentaries from semantically annotated media repositories
[2006-28] Borkur Sigurbjornsson (UVA) Focused Information Access using XML Element Retrieval

2007
[2007-01] Kees Leune (UvT) Access Control and Service-Oriented Architectures
[2007-02] Wouter Teepe (RUG) Reconciling Information Exchange and Confidentiality: A Formal Approach
[2007-03] Peter Mika (VU) Social Networks and the Semantic Web
[2007-04] Jurriaan van Diggelen (UU) Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach
[2007-05] Bart Schermer (UL) Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance
[2007-06] Gilad Mishne (UVA) Applied Text Analytics for Blogs
[2007-07] Natasa Jovanovic' (UT) To Whom It May Concern - Addressee Identification in Face-to-Face Meetings
[2007-08] Mark Hoogendoorn (VU) Modeling of Change in Multi-Agent Organizations
[2007-09] David Mobach (VU) Agent-Based Mediated Service Negotiation
[2007-10] Huib Aldewereld (UU) Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols
[2007-11] Natalia Stash (TUE) Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System
[2007-12] Marcel van Gerven (RUN) Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty
[2007-13] Rutger Rienks (UT) Meetings in Smart Environments; Implications of Progressing Technology
[2007-14] Niek Bergboer (UM) Context-Based Image Analysis
[2007-15] Joyca Lacroix (UM) NIM: a Situated Computational Memory Model
[2007-16] Davide Grossi (UU) Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems
[2007-17] Theodore Charitos (UU) Reasoning with Dynamic Networks in Practice
[2007-18] Bart Orriens (UvT) On the development an management of adaptive business collaborations
[2007-19] David Levy (UM) Intimate relationships with artificial partners
[2007-20] Slinger Jansen (UU) Customer Configuration Updating in a Software Supply Network

[2007-21] Karianne Vermaas (UU) Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005
[2007-22] Zlatko Zlatev (UT) Goal-oriented design of value and process models from patterns
[2007-23] Peter Barna (TUE) Specification of Application Logic in Web Information Systems
[2007-24] Georgina Ramírez Camps (CWI) Structural Features in XML Retrieval
[2007-25] Joost Schalken (VU) Empirical Investigations in Software Process Improvement

2008
[2008-01] Katalin Boer-Sorbán (EUR) Agent-Based Simulation of Financial Markets: A modular, continuous-time approach
[2008-02] Alexei Sharpanskykh (VU) On Computer-Aided Methods for Modeling and Analysis of Organizations
[2008-03] Vera Hollink (UVA) Optimizing hierarchical menus: a usage-based approach
[2008-04] Ander de Keijzer (UT) Management of Uncertain Data - towards unattended integration
[2008-05] Bela Mutschler (UT) Modeling and simulating causal dependencies on process-aware information systems from a cost perspective
[2008-06] Arjen Hommersom (RUN) On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective
[2008-07] Peter van Rosmalen (OU) Supporting the tutor in the design and support of adaptive e-learning
[2008-08] Janneke Bolt (UU) Bayesian Networks: Aspects of Approximate Inference
[2008-09] Christof van Nimwegen (UU) The paradox of the guided user: assistance can be counter-effective
[2008-10] Wauter Bosma (UT) Discourse oriented summarization
[2008-11] Vera Kartseva (VU) Designing Controls for Network Organizations: A Value-Based Approach
[2008-12] Jozsef Farkas (RUN) A Semiotically Oriented Cognitive Model of Knowledge Representation
[2008-13] Caterina Carraciolo (UVA) Topic Driven Access to Scientific Handbooks
[2008-14] Arthur van Bunningen (UT) Context-Aware Querying; Better Answers with Less Effort
[2008-15] Martijn van Otterlo (UT) The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.
[2008-16] Henriette van Vugt (VU) Embodied agents from a user's perspective

[2008-17] Martin Op 't Land (TUD) Applying Architecture and Ontology to the Splitting and Allying of Enterprises
[2008-18] Guido de Croon (UM) Adaptive Active Vision
[2008-19] Henning Rode (UT) From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search
[2008-20] Rex Arendsen (UVA) Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven.
[2008-21] Krisztian Balog (UVA) People Search in the Enterprise
[2008-22] Henk Koning (UU) Communication of IT-Architecture
[2008-23] Stefan Visscher (UU) Bayesian network models for the management of ventilator-associated pneumonia
[2008-24] Zharko Aleksovski (VU) Using background knowledge in ontology matching
[2008-25] Geert Jonker (UU) Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency
[2008-26] Marijn Huijbregts (UT) Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled
[2008-27] Hubert Vogten (OU) Design and Implementation Strategies for IMS Learning Design
[2008-28] Ildiko Flesch (RUN) On the Use of Independence Relations in Bayesian Networks
[2008-29] Dennis Reidsma (UT) Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans
[2008-30] Wouter van Atteveldt (VU) Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content
[2008-31] Loes Braun (UM) Pro-Active Medical Information Retrieval
[2008-32] Trung H. Bui (UT) Toward Affective Dialogue Management using Partially Observable Markov Decision Processes
[2008-33] Frank Terpstra (UVA) Scientific Workflow Design; theoretical and practical issues
[2008-34] Jeroen de Knijf (UU) Studies in Frequent Tree Mining
[2008-35] Ben Torben Nielsen (UvT) Dendritic morphologies: function shapes structure

2009

[2009-01] Rasa Jurgelenaite (RUN) Symmetric Causal Independence Models
[2009-02] Willem Robert van Hage (VU) Evaluating Ontology-Alignment Techniques
[2009-03] Hans Stol (UvT) A Framework for Evidence-based Policy Making Using IT
[2009-04] Josephine Nabukenya (RUN) Improving the Quality of Organisational Policy Making using Collaboration Engineering

[2009-05] Sietse Overbeek (RUN) Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality
[2009-06] Muhammad Subianto (UU) Understanding Classification
[2009-07] Ronald Poppe (UT) Discriminative Vision-Based Recovery and Recognition of Human Motion
[2009-08] Volker Nannen (VU) Evolutionary Agent-Based Policy Analysis in Dynamic Environments
[2009-09] Benjamin Kanagwa (RUN) Design, Discovery and Construction of Service-oriented Systems
[2009-10] Jan Wielemaker (UVA) Logic programming for knowledge-intensive interactive applications
[2009-11] Alexander Boer (UVA) Legal Theory, Sources of Law & the Semantic Web
[2009-12] Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin) Operating Guidelines for Services
[2009-13] Steven de Jong (UM) Fairness in Multi-Agent Systems
[2009-14] Maksym Korotkiy (VU) From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)
[2009-15] Rinke Hoekstra (UVA) Ontology Representation - Design Patterns and Ontologies that Make Sense
[2009-16] Fritz Reul (UvT) New Architectures in Computer Chess
[2009-17] Laurens van der Maaten (UvT) Feature Extraction from Visual Data
[2009-18] Fabian Groffen (CWI) Armada, An Evolving Database System
[2009-19] Valentin Robu (CWI) Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets
[2009-20] Bob van der Vecht (UU) Adjustable Autonomy: Controling Influences on Decision Making
[2009-21] Stijn Vanderlooy (UM) Ranking and Reliable Classification
[2009-22] Pavel Serdyukov (UT) Search For Expertise: Going beyond direct evidence
[2009-23] Peter Hofgesang (VU) Modelling Web Usage in a Changing Environment
[2009-24] Annerieke Heuvelink (VUA) Cognitive Models for Training Simulations
[2009-25] Alex van Ballegooij (CWI) "RAM: Array Database Management through Relational Mapping"
[2009-26] Fernando Koch (UU) An Agent-Based Model for the Development of Intelligent Mobile Services
[2009-27] Christian Glahn (OU) Contextual Support of social Engagement and Reflection on the Web
[2009-28] Sander Evers (UT) Sensor Data Management with Probabilistic Models
[2009-29] Stanislav Pokraev (UT) Model-Driven Semantic Integration of Service-Oriented Applications

Notes

ABOUT THE AUTHOR

Stanislav Pokraev received a master's degree (MSc) in Electronics and Automatics from the Technical University of Sofia, Bulgaria.

In 1998, Stanislav joined the Computer Science department of the University of Twente as a research fellow. Here, he carried out research in the area of enterprise modelling and telelearning.

From 1999 to 2001, Stanislav worked as a research engineer at KPN Research. His work at KPN involved the design and development of distributed information systems.

Since 2001, Stanislav has been a researcher and consultant at Novay. As a member of the Service Architectures group, he investigates how model-driven design approaches can be used to integrate service-oriented applications.

Stanislav has authored and co-authored over twenty international publications, including workshop and conference papers, journal articles, and book chapters. He has played a leading role in the design and the implementation of various prototypes and demonstrators. Stanislav has also served as a reviewer for international journals and conferences.

Stanislav Pokraev

MODEL-DRIVEN SEMANTIC INTEGRATION OF SERVICE-ORIENTED APPLICATIONS

Stanislav Pokraev

The integration of enterprise applications is an extremely complex problem since most applications have not been designed to work with other applications. That is, they have different information models, do not share a common state, and do not consult each other when updating their states. Unfortunately, domain experts, who have the required knowledge to resolve the mismatches between the applications, are typically not IT experts. Their expertise is only used in the very early stages of integration projects, namely, in the requirements elicitation stage. This makes the gap between business integration requirements and the software implementation of the integration solution wide, which, in turn, additionally complicates the integration process and often leads to the realization of solutions that work incorrectly.

The work presented in this thesis makes contributions in the area of Enterprise Application Integration. We propose a method for the semantic integration of service-oriented applications. The key feature of the method is that semantically-rich service models, at different levels of abstraction, are employed to build flexible integration solutions. The method allows system integrators (both domain experts and software developers) to address only a limited set of concerns in a series of design steps. In addition, the proposed method supports handling of changes in the implementation technology and integration requirements. Finally, integration solutions produced by the method can be formally verified for correctness using automatic reasoners.

ISBN: 978-90-75176-49-0

novay